ESCUELA SUPERIOR DE INGENIERÍA

INGENIERÍA INFORMÁTICA

Generador de casos de prueba basado en estrategias personalizables

Miguel Ángel Pérez Montero

18 de julio de 2013

ESCUELA SUPERIOR DE INGENIERÍA

INGENIERO INFORMÁTICO

Generador de casos de prueba basado en estrategias personalizables

- Departamento: Ingeniería Informática.
- Directores del proyecto: Juan José Dominguéz Jiménez, Antonio García Dominguez.
- Autor del proyecto: Miguel Ángel Pérez Montero.

Cádiz, 18 de julio de 2013

Fdo: Miguel Ángel Pérez Montero

Licencia

Notación y formato

En esta sección, incluiremos los aspectos relevantes a la notación y el formato a lo largo del documento. Dicha notación es la siguiente:

Si nos vamos a referir a un directorio en particular, usaremos la notación:

/home

Cuando nos refiramos a un programa en concreto, utilizaremos la notación:

emacs.

Cuando nos refiramos a un comando, o función de un lenguaje, usaremos la notación:

`quicksort`.

Si nos vamos a referir a una clase, utilizaremos la notación:

MiClase

Índice general

1. Motivación y contexto **1**

1.1. Introducción 1

1.2. Objetivos 2

1.3. Alcance 3

1.4. Lenguajes específicos del dominio 3

2. Tecnologías **7**

2.1. WS-BPEL 7

2.2. WSDL 8

2.3. SOAP 13

2.4. XML Schema 14

2.5. BPELUnit 14

2.6. Apache Velocity 18

2.7. CSV 21

3. Planificación **23**

3.1. Etapas 23

3.1.1. Elicitación de requisitos 24

3.1.2. Estudio de tecnologías 24

3.1.3. Plugin con *Xtext* 24

3.1.4. Plugin de ejecución 24
3.1.5. Diseño de la estrategia 25
3.1.6. Cambio de la gramática 25
3.1.7. Cambio del analizador 25
3.1.8. Implementación de estrategias 25
3.1.9. Pruebas unitarias 25
3.1.10. Documentación 26
3.1.11. Paquete de instalación y líneas de ordenes 26
3.2. Diagrama Gantt 26

4. Análisis **31**
4.1. Requisitos funcionales 31
4.2. Estudio del mercado 33
4.3. Requisitos de implementación 36
4.4. Atributos del sistema 36
4.5. Casos de uso 37
4.5.1. Generar datos aleatorios 37
4.5.2. Mostrar ayuda 38
4.5.3. Abrir fichero 38
4.5.4. Cerrar fichero 39
4.5.5. Guardar fichero 39
4.5.6. Editar fichero 40
4.5.7. Detectar errores de formato 40
4.5.8. Navegar por el fichero 41

5. Diseño **43**
5.1. Arquitectura general del sistema 43
5.2. Sistema de tipos de TestGenerator 44
5.3. Analizador de catálogos de ServiceAnalyzer 51

5.4. Analizador de especificaciones TestSpec 51
5.4.1. Motivación 51
5.4.2. Sintaxis abstracta 51
5.4.3. Sintaxis concreta 52
5.4.4. Ejemplos 55
5.5. Estrategias básicas 57
5.5.1. Estrategias aleatorias 57
5.5.2. Estrategias combinatorias 59
5.6. Estrategias compuestas 61
5.7. Formateadores 64

6. Implementación y pruebas **67**
6.1. Integración continua 68
6.1.1. Jenkins 68
6.1.2. Subversion 68
6.1.3. Maven 3 69
6.1.4. Tycho 70
6.1.5. Nexus 71
6.1.6. Sonar 71
6.2. Implementación de los analizadores 72
6.2.1. Analizador de los ficheros TestSpec 72
6.3. Implementación del editor 75
6.4. Generadores de datos aleatorios 78
6.5. Estrategias de generación 78
6.5.1. Distribuciones estadísticas 78
6.5.2. Estrategias combinatorias 79
6.6. Representación de los tipos de datos 80
6.6.1. Enteros 80
6.6.2. Números en coma flotante 83
6.6.3. Cadenas 84

6.6.4. Fechas 84
6.6.5. Contenedores 86
6.7. Pruebas 86
6.7.1. Naturaleza de las pruebas 87
6.7.2. Diseño de las pruebas 88
6.8. Validación 90

7. Conclusiones **93**
7.1. Valoración personal 93
7.2. Trabajo futuro 94

A. Manual de usuario **97**
A.1. Instalación de TestGenerator 97
A.2. Uso de la herramienta 99
A.2.1. Terminal 99
A.2.2. Plugin de Eclipse 100
A.3. Creación de ficheros TestSpec 104
A.3.1. Sentencias «typedef» 105
A.3.2. Sentencias «declaration» 106
A.3.3. Estrategias «strategy» 107
A.4. Ejemplo de un fichero *spec* 107

B. Manual de desarrollador **109**
B.1. Requisitos del sistema 109
B.2. Obtención del código 109

Bibliografia y referencias **112**

GNU Free Documentation License **117**
1. APPLICABILITY AND DEFINITIONS 118
2. VERBATIM COPYING 120

3. COPYING IN QUANTITY 120
4. MODIFICATIONS 121
5. COMBINING DOCUMENTS 124
6. COLLECTIONS OF DOCUMENTS 124
7. AGGREGATION WITH INDEPENDENT WORKS 125
8. TRANSLATION 125
9. TERMINATION 126
10. FUTURE REVISIONS OF THIS LICENSE 126
11. RELICENSING 127
ADDENDUM: How to use this License for your documents 128

Indice de figuras

3.1. Diagrama de Gantt I 27
3.2. Diagrama de Gantt II 28

4.1. Diagrama de casos de uso 37

5.1. Diagrama de los paquetes 45
5.2. Estructura del sistema 46
5.3. Diagrama de los tipos I 49
5.4. Diagrama de los tipos II 50
5.5. Metamodelo de TestSpec 53
5.6. Ejemplo visual del algoritmo Comb 61
5.7. Diagrama del analizador 64
5.8. Diagrama del formateador 66

6.1. Diagrama de la estructura de directorios 77
6.2. Ejemplo de automata finito generado por Xeger para la expresión regular a[aeo]* 85

A.1. Instalación de los plugins 99
A.2. Ejemplo autocompletado 101
A.3. Corrección de errores 101

A.4. Menu outline 102
A.5. Instalación de los plugins 103
A.6. Ventana de generación 103
A.7. Éxito en la generación 104
A.8. Resultado de la generación 104

Indice de listados

2.1. Ejemplo de WS-BPEL 9
2.2. Ejemplo de WSDL 11
2.3. Ejemplo de SOAP 13
2.4. Ejemplo de XML Schema 14
2.5. Ejemplo de un fichero BPTS 15
2.6. Ejemplo de una plantilla Velocity 19
2.7. Salida producida por la plantilla Velocity 20
2.8. Ejemplo de plantilla Velocity para generar un mensaje SOAP 20
2.9. Ejemplo de un fichero CSV 21

4.1. Ejemplo de uso de *JCheck* 34
4.2. Ejemplo de uso de *QuickCheck for Java* 35

5.1. Gramática del lenguaje TestSpec 52
5.2. Ejemplo de especificación TestSpec 56
5.3. Ejemplo complejo de especificación TestSpec 58
5.4. Ejemplo de sintaxis de estrategias 60
5.5. Ejemplo de necesidad de ámbitos 62
5.6. Ejemplo de ámbitos 63
5.7. Ejemplo de tipos con ámbitos 63

5.8. Ejemplo de fichero de datos CSV generado por TestGenerator 65
5.9. Ejemplo de fichero de datos Apache Velocity generado por TestGenerator . 65

6.1. Gramática del lenguaje TestSpec 73
6.2. Comprobación semántica del editor para TestSpec 76
6.3. Algoritmo Comb 81

A.1. Ejemplo de especificación TestSpec 108

Capítulo 1

Motivación y contexto

1.1. Introducción

Dentro de la línea de pruebas del software del grupo de investigación UCASE de la Universidad de Cádiz, se trabaja en la mejora de conjuntos de casos de prueba para composiciones de servicios Web escritas en WS-BPEL. Para mejorar el conjunto de casos de prueba, el grupo de investigación propone localizar aquellos posibles fallos que el conjunto de casos de partida no detectaría, y extenderlo para que sí los detecte. Actualmente, el grupo ha obtenido buenos resultados localizando los fallos usando análisis de mutaciones (con sus herramientas *MuBPEL* [1] y *GAmera* [2]), y ha comenzado a trabajar en el área de generación de casos de prueba. Sin embargo, extender un conjunto de casos de prueba para una composición WS-BPEL es difícil, ya que hay que tener en cuenta numerosas tecnologías.

En un Proyecto Fin de Carrera anterior, "Generador de casos de prueba aleatorio basado en especificaciones abstractas", se implementó *TestGenerator* [3] una aplicación que es capaz de generar los datos necesarios de forma aleatoria con unas restricciones específicas dadas en un fichero de entrada al programa. Ante el éxito de este programa, algunos formas de adopción del proyecto son: como parte de Rodan (tesis de Antonia Estero) y de SODM+T (tesis de Antonio García) y ha sido extendido por otros miembros del grupo en otras direcciones, se desea realizar mejoras al mismo tanto en utilidad

como en facilidad de uso.

1.2. Objetivos

En *TestGenerator* se tuvo que diseñar un lenguaje de dominio específico (§1.4) denominado *TestSpec* cuyos ficheros tienen la extensión spec. Este tipo de fichero han tenido un gran éxito dentro del grupo UCASE, pero escribir un fichero con dicho lenguaje puede ser algo incómodo puesto que no existe ningún editor para poder escribir en dicho lenguaje. Como mejora se realizará un editor para poder escribir ficheros spec de manera más fácil y cómoda. Dicho editor tendrá las siguientes utilidades:

- Resalto de palabras reservadas.
- Autocompletado del texto.
- Detector de errores de sintaxis.
- Detector de errores semánticos.

En definitiva, queremos desarrollar un editor con todas las comidades que existen hoy en dia a la hora de escribir en un determinado lenguaje de programación como las que nos ofrece *NetBeans* o *Eclipse*.

Por otro lado se desea mejorar el núcleo de la aplicación: el generador de datos. Los tipos de valores y restricciones a tener en cuenta en este generador serán los mismos que se implementaron en el proyecto anterior; a modo de recordatorio estos tipos con sus restricciones son los siguientes: enteros, números reales, cadenas, fechas, horas, duraciones, listas homogéneas y tuplas. Como restricciones, se incluirán valores y longitudes mínimas y máximas, enumeraciones de valores válidos y expresiones regulares a cumplir por las cadenas, entre otras.

En dicho núcleo, se incluirá un concepto nuevo llamado estrategia de generación. Con este concepto representaremos de qué forma vamos a generar los datos. Las distribuciones de probabilidad a implementar serán las siguientes: Gaussiana, Exponencial, Binomial y Poisson. También se implementarán estrategias combinatorias. Como resultado

del nuevo concepto de estrategia habrá que modificar el DSL para incluir dicho concepto, pero deberá ser un superconjunto del anterior. Esto implica modificación completa del analizador de los ficheros *spec* y generador de datos.

1.3. Alcance

Este proyecto será capaz de dar respaldo al sistema de tipos que implementa *ServiceAnalyzer*. Este dominio cubre las necesidades del grupo UCASE, ya que fue escogido en base a las composiciones WS-BPEL que se están estudiando actualmente.

En este Proyecto Fin de Carrera, se implementará distintas estrategias de generación y deberá tener la estructura necesaria para poder seguir añadiendo más estrategias sin que esto sea una labor muy tediosa. Por restricciones de tiempo las estrategias de generación seguirán las distribuciones de probabilidad gaussiana, exponencial, binomial y poisson, además de algunas estrategias combinatorias.

La comunicación entre el usuario y la herramienta será a través de la línea de órdenes y a partir del edtior.

1.4. Lenguajes específicos del dominio

Los lenguajes específicos del dominio [5] (domain-specific language - DSL) son lenguajes destinados a una única tarea concisa y con un ámbito muy particular, es decir, es un lenguaje dedicado a un problema de dominio en particular, o una técnica de representación o resolución de problemas específica. Este concepto no es nuevo, ya que desde siempre existieron lenguajes de programación de propósito específico. Algunos de ellos muy conocidos, unos ejemplos podrían ser los siguientes:

- Apache Maven
- Structured Query Language (SQL)
- Flex

- Bison
- HTML
- CSS

En el lado opuesto a los DSL encontramos los lenguajes de proposito general como por ejemplo Java o C y los lenguajes de modelado de propósito general, como UML. Los DSL se crean específicamente para resolver problemas dentro de un dominio en particular, y no están pensados para resolver problemas fuera de este dominio. En cambio, los lenguajes de propósito general se crean para resolver probelmas en muchos dominios. Un dominio también puede ser un área de negocio específica.

Una forma de clasificar los DSL puede entre los DSL internos y externos. Los DSL internos son formas particulares de la utilización de un lenguaje principal para dar la lengua "host" la idea de un lenguaje particular. Este enfoque ha sido popularizada recientemente por la comunidad Ruby aunque ha tenido una larga tradición en otros idiomas - especialmente Lisp. Aunque por lo general es más fácil en los lenguajes de bajo nivel, otros ejemplo en los que se pueden desarrollar DSL internos son en lenguajes tradicionales como Java y C#. A los DSL internos también se les conoce como DSL integrados.

Los DSL externos tienen su propia sintaxis personalizada y se debe escribir un analizador completo para poder procesarlos. Hay una tradición muy fuerte de hacer esto en la comunidad Unix. Muchas configuraciones XML han terminado como DSL externos, aunque la sintaxis de XML está muy adecuado para este fin.

Los DSLs más comunes hoy en día en la naturaleza son de texto, pero existen también otra forma de representación, los DSL gráficos. DSLs gráfico necesitan de una herramienta para poder trabajar con ellos, es decir, un editor propio. Este tipo de editores son menos comunes, pero muchas personas piensan que tienen el potencial de mejorar profundamente nuestra forma de hacer la programación.

Los DSL pueden ser ejecutados bien por la interpretación o la generación de código. Interpretación (leer el script DSL y ejecutarlo en tiempo de ejecución) suele ser más fácil, pero la generación de código a veces es esencial. Por lo general, el código generado es

en sí un lenguaje de alto nivel, como Java o C.

Entre las ventajas de los DSL podemos mencionar:

- Los DSL permiten expresar soluciones usando los términos y el nivel de abstracción apropiado para el dominio del problema. En consecuencia, los mismos expertos de dominio pueden comprender, validar, modificar y a menudo desarrollar programas en DSL.
- Los DSL mejoran la calidad, productividad, confianza, mantenibilidad, portabilidad y reusabilidad de las aplicaciones.
- Los DSL permiten validaciones a nivel del dominio. Mientras las contrucciones del lenguaje estén correctas, cualquier sentencia escrita puede considerarse correcta.

Alguna de las desventajas de los DSL son:

- El costo de aprender un nuevo lenguaje frente a su aplicación limitada.
- El costo de diseñar, implementar y mantener un DSL y las herramientas para trabajar con él.
- Encontrar, establecer y mantener el alcance adecuado.
- Dificultad para balancear las ventajas y desventajas entre las construcciones de los DSL y de los lenguajes de propósito general.
- Potencial pérdida de eficiencia y rendimiento en comparación con el software escrito “a mano”.

Capítulo 2

Tecnologías

Para poder entender bien cómo encaja este proyecto en el grupo de investigación UCASE es necesario entender algunas de las tecnologías que utiliza dicho grupo, puesto que de forma directa o indirecta afecta a los requisitos de la aplicación así como a su desarrollo.

2.1. WS-BPEL

WS-BPEL (Business Process Execution Language) es el lenguaje en el que se centran algunas de las líneas de trabajo del grupo UCASE. WS-BPEL es un lenguaje para la composición de servicios Web. Está basado en XML y sirve para el control centralizado de la invocación de diferentes servicios Web, con cierta lógica de negocio añadida que ayuda a la programación en gran escala.

La estructura de un proceso WS-BPEL se divide en cuatro secciones:

1. Definición de relaciones con los socios externos, que son el cliente que utiliza el proceso de negocio y los WS a los que llama el proceso.

2. Definición de las variables que emplea el proceso.

3. Definición de los distintos tipos de manejadores que puede utilizar el proceso. Pueden definirse manejadores de fallos, que indican las acciones a realizar en caso

de producirse un fallo interno o en un WS al que se llama. También se definen los manejadores de eventos, que especifican las acciones a realizar en caso de que el proceso reciba una petición durante su flujo normal de ejecución.

4. Descripción del comportamiento del proceso de negocio; esto se logra a través de las actividades que proporciona el lenguaje.

Todos los elementos definidos anteriormente son globales si se declaran dentro del proceso. Sin embargo, también existe la posibilidad de declararlos de forma local mediante el contenedor scope, que permite dividir el proceso de negocio en diferentes ámbitos. Los principales elementos constructivos de un proceso WS-BPEL son las actividades, que pueden ser de dos tipos: básicas y estructuradas. Las actividades básicas son las que realizan una determinada labor (recepción de un mensaje, manipulación de datos, etc.). Las actividades estructuradas pueden contener otras actividades y definen la lógica de negocio. A las actividades pueden asociarse un conjunto de atributos y un conjunto de contenedores. Estos últimos pueden incluir diferentes elementos, que a su vez pueden tener atributos asociados. En el listado 2.1 podemos ver un ejemplo explicativo.

2.2. WSDL

WSDL (Web Services Description Language) es un formato XML utilizado para describir la interfaz pública de servicios Web. Una composición WS-BPEL normalmente reúne varios servicios Web (descritos con WSDL) en uno de nivel superior (la composición), también descrito con WSDL.

WSDL consta de un conjunto muy amplio de reglas y normas. En la práctica, no se suelen implementar todas: hacerlo sería muy complejo y costoso. La mayoría de las implementaciones actuales se centran en el subconjunto definido por la especificación WS-I Basic Profile 1.1 [4]. Este subconjunto permite conseguir una mayor interoperabilidad entre todas las implementaciones existentes.

Está basado en XML y describe la forma de comunicación, es decir, los requisitos del protocolo y los formatos de los mensajes necesarios para interactuar con los servicios

Listado 2.1: Ejemplo de WS-BPEL

```
<flow>  ← Actividad estructurada
  <links>  ← Contenedor
    <link name="comprobarVuelo-reservarVuelo" ←Atributo/> ←Elemento
  </links>
  <invoke name="comprobarVuelo" . . . > ←Actividad basica
    <sources> ← Contenedor
      <source linkName="comprobarVuelo-reservarVuelo" ←Atributo/> ←Elemento
        </sources>
  </invoke>
  <invoke name="comprobarHotel" . . . />
  <invoke name="comprobarAlquilerCoche" . . . />
  <invoke name="reservarVuelo" . . . >
  <targets> ← Contenedor
    <target linkName="comprobarVuelo-reservarVuelo" /> ←Elemento
  </targets>
  </invoke>
</flow>
```

listados. Las operaciones y mensajes que utilizan los servicios se describen en abstracto y se ligan después al protocolo concreto de red y al formato del mensaje. Así, WSDL se usa a menudo en combinación con SOAP (§ 2.3) y XML Schema (§ 2.4).

Un programa cliente que se conecta a un servicio web puede leer el WSDL para determinar qué funciones están disponibles en el servidor. Los tipos de datos se describen en el archivo WSDL usando XML Schema. El cliente puede usar distintas tecnologías, en el grupo UCASE la elegida es SOAP para hacer la llamada a una de las funciones listadas en el WSDL. El WSDL nos permite tener una descripción de un servicio web. Especifica la interfaz abstracta a través de la cual un cliente puede acceder al servicio y los detalles de cómo se debe utilizar.

Podemos ver un ejemplo en el listado 2.2 donde la estructura WSDL tiene los siguientes elementos:

- Tipos de datos `<types>`: Esta sección define los tipos de datos usados en los mensajes. Se utilizan los tipos definidos en la especificación de esquemas XML.
- Mensajes `<message>`: Aquí definimos los elementos de mensaje. Cada mensaje puede consistir en una serie de partes lógicas. Las partes pueden ser de cualquiera de los tipos definidos en la sección anterior.
- Tipos de puerto `<portType>`: Con este apartado definimos las operaciones permitidas y los mensajes intercambiados en el servicio.
- «Bindings» `<binding>`: Especificamos las definiciones de los protocolos de comunicación usados entre los servicio. El elemento `binding` está formado por dos atributos `name` y `type`, el primer define el nombre de la unión y el segundo especifica el protocolo usado.
- Servicios <service>: Conjunto de puertos y dirección de los mismos. Esta parte final hace referencia a lo aportado por las secciones anteriores.

Con estos elementos no sabemos qué hace un servicio pero sí disponemos de la información necesaria para interactuar con él.

Listado 2.2: Ejemplo de WSDL

```
<definitions name="StockQuote"
         targetNamespace="http://example.com/stockquote.wsdl"
         xmlns:tns="http://example.com/stockquote.wsdl"
         xmlns:xsd1="http://example.com/stockquote.xsd"
         xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
         xmlns="http://schemas.xmlsoap.org/wsdl/">

   <types>
      <schema targetNamespace="http://example.com/stockquote.xsd"
            xmlns="http://www.w3.org/2000/10/XMLSchema">
         <element name="TradePriceRequest">
            <complexType>
                <all>
                    <element name="tickerSymbol" type="string"/>
                </all>
            </complexType>
         </element>
         <element name="TradePrice">
            <complexType>
                <all>
                    <element name="price" type="float"/>
                </all>
            </complexType>
         </element>
      </schema>
   </types>

   <message name="GetLastTradePriceInput">
       <part name="body" element="xsd1:TradePriceRequest"/>
   </message>

   <message name="GetLastTradePriceOutput">
```

```
33        <part name="body" element="xsd1:TradePrice"/>
34    </message>
35
36    <portType name="StockQuotePortType">
37        <operation name="GetLastTradePrice">
38           <input message="tns:GetLastTradePriceInput"/>
39           <output message="tns:GetLastTradePriceOutput"/>
40        </operation>
41    </portType>
42
43    <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
44        <soap:binding style="document"
45                                transport="http://schemas.xmlsoap.org/soap/http"/>
46        <operation name="GetLastTradePrice">
47           <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
48           <input>
49               <soap:body use="literal"/>
50           </input>
51           <output>
52               <soap:body use="literal"/>
53           </output>
54        </operation>
55    </binding>
56
57    <service name="StockQuoteService">
58        <documentation>My first service</documentation>
59        <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
60           <soap:address location="http://example.com/stockquote"/>
61        </port>
62        </service>
63
64 </definitions>
```

Listado 2.3: Ejemplo de SOAP

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

       <soap:Header>
              ...
       </soap:Header>

       <soap:Body>
              ...
              <soap:Fault>
                     ...
              </soap:Fault>
       </soap:Body>

</soap:Envelope>
```

2.3. SOAP

SOAP (Simple Object Access Protocol) es un protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML. Este protocolo deriva de un protocolo creado por David Winer en 1998, llamado XML-RPC. SOAP fue creado por Microsoft, IBM y otros y está actualmente bajo el auspicio de la W3C. Es uno de los protocolos utilizados en los servicios Web. En el listado 2.3 podemos ver el esqueleto de un mensaje SOAP.

Listado 2.4: Ejemplo de XML Schema

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

2.4. XML Schema

XML Schema es un lenguaje utilizado para describir la estructura y las restricciones de los contenidos de los documentos XML de una forma muy precisa, más allá de las normas sintácticas impuestas por XML. Fue desarrollado por el World Wide Web Consortium (W3C) y alcanzó el nivel de recomendación en mayo de 2001. Podemos ver un ejemplo en el listado 2.4 donde podemos ver que está compuesto por un elemento llamado «note» que a su vez está definido por cuatro elementos de tipo cadena denominadas «to», «from», «heading» y «body».

2.5. BPELUnit

BPELUnit es una biblioteca de pruebas unitarias para composiciones WS-BPEL, creada por Philip Mayer. Puede usar cualquier motor que implemente WS-BPEL 2.0. Entre sus

principales características está el uso de un formato XML (BPELUnit Test Specification o BPTS) para describir los casos de prueba a ejecutar y la posibilidad de sustituir servicios externos con otros servicios («mockups») que los simulen desarrollando el comportamiento indicado en la especificación proporcionada por el usuario. Además, BPELUnit ofrece posibilidades para añadir envíos síncronos y asíncronos. Podemos ver un ejemplo en el listado 2.5.

A partir de la versión 1.5 de *BPELUnit*, los ficheros BPTS incorporan el lenguaje de plantillas Apache Velocity. Las plantillas permiten generar los mensajes a partir de una fuente de datos y una serie de variables predefinidas. Esto permite que el usuario pueda definir fácilmente diversos casos de prueba que tengan las mismas actividades, pero distinto contenido en los mensajes. Con ello, se obtienen, principalmente, tres ventajas:

- Facilita la generación de los casos de prueba y se hace más sencilla su automatización.
- Se ha separado la generación de casos de prueba de los detalles de WSDL y SOAP.
- Permite la creación de «mockups» más inteligentes, que consideren los mensajes que reciben (qué hay en la petición).

Listado 2.5: Ejemplo de un fichero BPTS

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <tes:testSuite
3    xmlns:esq="http://xml.netbeans.org/schema/Loans"
4    xmlns:ap="http://j2ee.netbeans.org/wsdl/ApprovalService"
5    xmlns:as="http://j2ee.netbeans.org/wsdl/AssessorService"
6    xmlns:sp="http://j2ee.netbeans.org/wsdl/LoanService"
7    xmlns:pr="http://enterprise.netbeans.org/bpel/N6 ServicioPrestamo/
8              LoanApprovalProcess"
9    xmlns:tes="http://www.bpelunit.org/schema/testSuite">
10
11   <tes:name>LoanServiceTest</tes:name>
```

```
12  <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
13
14  <tes:deployment>
15    <tes:put name="LoanApprovalProcess" type="activebpel">
16      <tes:wsdl>LoanService.wsdl</tes:wsdl>
17      <tes:property name="BPRFile">LoanApprovalDoc.bpr</tes:property>
18    </tes:put>
19    <tes:partner name="assessor" wsdl="AssessorService.wsdl"/>
20    <tes:partner name="approver" wsdl="ApprovalService.wsdl"/>
21  </tes:deployment>
22
23  <tes:testCases>
24    <tes:testCase name="LargeAmount" basedOn="" abstract="false" vary="false">
25      <tes:clientTrack>
26        <tes:sendReceive
27           service="sp:LoanServiceService"
28           port="LoanServicePort"
29           operation="grantLoan">
30
31          <tes:send fault="false">
32            <tes:data>
33              <esq:ApprovalRequest>
34                <esq:amount>150000</esq:amount>
35              </esq:ApprovalRequest>
36            </tes:data>
37          </tes:send>
38
39          <tes:receive fault="false">
40            <tes:condition>
41              <tes:expression>esq:ApprovalResponse/esq:accept</tes:expression>
42              <tes:value>'true'</tes:value>
43            </tes:condition>
44          </tes:receive>
```

```
45
46        </tes:sendReceive>
47      </tes:clientTrack>
48
49      <tes:partnerTrack name="approver">
50        <tes:receiveSend
51           service="ap:ApprovalServiceService"
52           port="ApprovalServicePort"
53           operation="approveLoan">
54
55          <tes:send fault="false">
56            <tes:data>
57              <esq:ApprovalResponse>
58                <esq:accept>true</esq:accept>
59              </esq:ApprovalResponse>
60            </tes:data>
61          </tes:send>
62
63          <tes:receive fault="false"/>
64        </tes:receiveSend>
65      </tes:partnerTrack>
66
67      <tes:partnerTrack name="assessor"/>
68    </tes:testCase>
69    ...
70  </tes:testCases>
71 </tes:testSuite>
```

En el listado 2.5 podemos observar la estructura de un fichero BPTS mediante un fragmento de un fichero de pruebas de la clásica composición WS-BPEL de aprobación de un préstamo («LoanApproval»). Esta composición recibe un mensaje de un cliente que solicita una cierta cantidad de dinero. Dependiendo de la cantidad solicitada, el proceso WS-BPEL invoca al WS asesor («asessor») cuando la cantidad que se solicita es

menor o igual a 10000, o al WS aprobador («approver»), en otro caso. La salida del WS asesor se corresponde con el nivel de riesgo del cliente. Si el riesgo es bajo, se concede el préstamo, en caso contrario la petición se envía al WS aprobador, el cuál toma la decisión final de aceptación del préstamo.

Hasta la línea 8 tenemos el elemento raíz y la definición de los espacios de nombres XML. En este caso, el prefijo `tes` está asociado al espacio de nombres de BPELUnit y el resto, son prefijos propios de esta composición y el de los envoltorios SOAP.

A continuación encontramos la sección de despliegue (líneas 10–20). En ella podemos comprobar que la composición se comunica con dos «mockups»: el asesor y el aprobador.

Finalmente, de la línea 22 en adelante, aparece la sección de los casos de prueba en la que debe ir incluido cada bloque `testCase`. En concreto, se ha incluido un caso de prueba de ejemplo en el que el cliente solicita un préstamo por valor de 150.000€ (líneas 30–36), que es aceptado directamente por el aprobador (líneas 52–58), sin que tenga que intervenir el asesor (línea 63). Finalmente, el aprobador comunica al cliente su decisión (líneas 38–44).

2.6. Apache Velocity

Apache Velocity es un motor de plantillas basado en Java. Le permite a los diseñadores de páginas hacer referencia a métodos definidos dentro del código Java. Los diseñadores Web pueden trabajar en paralelo con los programadores Java para desarrollar sitios de acuerdo al modelo de Modelo-Vista-Controlador (MVC), permitiendo que los diseñadores se concentren únicamente en crear un sitio bien diseñado y que los programadores se encarguen solamente de escribir código de primera calidad. Velocity separa el código Java de las páginas Web, haciendo el sitio más mantenible a largo plazo y presentando una alternativa viable a Java Server Pages (JSP) o PHP.

BPELUnit utiliza Velocity de dos formas: como lenguaje de plantillas propiamente dicho para generar los mensajes a enviar a la composición WS-BPEL, y como formato de entrada para recibir los valores de las variables a usar en las plantillas de los mensajes.

Listado 2.6: Ejemplo de una plantilla Velocity

```
<html>
       <body>
              <h1 > $titulo </h1>
              <br/>
              tabla:
              <table>
              #foreach ($alumno in $listado)
                     <tr><td bgcolor="$color">$alumno</td></tr>
              #end
              <table/>
       </body>
</html>
```

Este PFC se centra en el segundo uso.

En el listado 2.6 podemos ver un ejemplo sencillo de un fichero *html* con código Velocity, dónde podemos observar que existe una variable para definir el contenido de `h1` y un bucle para crear una tabla con los alumnos. Si dicho fichero se ejecutara asignando "Tabla de estudiantes" a `titulo`, "white" a `color` y una lista con "Pedro", "Ana" y "Juan" a `listado`, la salida resultante de dicho ejemplo es la mostrada en 2.7.

Para que quede más claro, también vamos a mostrar un ejemplo más parecido a los que va a tratar este PFC. Puede verse un ejemplo de plantilla Velocity en el listado 2.8.

Los ficheros Velocity están formados por filas, cada fila representa los valores de cada una de las variables para cada caso de prueba. La primera componente corresponderá al primer caso de prueba, la segunda al segundo, etc.

Las variables que se vayan a declarar en la fuente de datos cuando el tipo es Velocity obligatoriamente han de listarse (separadas por espacios) en la propiedad iteratedVars. Todas las variables que no se declaren aquí pero sí aparezcan en la fuente de datos, únicamente se copiaran tal cual, pero no se sustituirán en las plantillas.

Todas las variables incluidas en `iteratedVars` deben tener asociadas en la fuente de

Listado 2.7: Salida producida por la plantilla Velocity

```
<html>
       <body>
              <h1 > Tabla de Estudiantes </h1>
              <br/>
              tabla:
              <br/>
              <table/>
                     <tr><td bgcolor="white">Pedro</td></tr>
                     <tr><td bgcolor="white">Juan</td></tr>
                     <tr><td bgcolor="white">Ana</td></tr>
              <table/>
              <br/>
   </body>
</html>
```

datos una lista de valores con idéntico número de elementos.

Listado 2.8: Ejemplo de plantilla Velocity para generar un mensaje SOAP

```
<bpelunit:send fault="false">
  <bpelunit:template>
    <srv:ApprovalResponse>
      #set( $sAmount = $xpath.evaluateAsString("//srv:amount", $request) )
      #set( $amount = $integer.parseInt($sAmount) )
      #if( $ap_reply != 'smart' )
      <srv:accept>$ap_reply</srv:accept>
      #elseif( $ap_limit > $amount )
      <srv:accept>true</srv:accept>
      #else
      <srv:accept>false</srv:accept>
      #end
    </srv:ApprovalResponse>
```

```
  </bpelunit:template>
</bpelunit:send>
```

2.7. CSV

CSV («comma-separated values» o «valores separados por comas») describe a un conjunto de formatos de fichero basados en texto plano donde cada línea representa una fila, y los valores de cada fila están separados por un marcador determinado. Normalmente, este marcador es una coma (de ahí su nombre).

CSV es otro formato de entrada que permite *BPELUnit* para sus plantillas. Puede verse un ejemplo en el listado 2.9. Cada fila proporciona los datos de un trayecto de tren: tipo de máquina, origen, destino, número de trenes y después las horas de salidas.

Listado 2.9: Ejemplo de un fichero CSV

```
tren,Origen,Destino,númeroTrenes,horasSalidas
pequenio,cádiz,sevilla,1,9:00,11:00,13:00,15:00
grande,cádiz,sevilla,1,17:00,19:00,19:00,21:00
grande,sevilla,huelva,1,9:00,11:00
ave,cádiz,sevilla,1,9:15,10:15
pequenio,sevilla,jerez,1,9:00,10:00,13:00,14:00
```

Capítulo 3

Planificación

En este capítulo veremos cómo se ha organizado temporalmente el trabajo de este proyecto así como las tareas que han tenido lugar y la planificación de las mismas.

El proyecto se ha desarrollado a lo largo de unos diez meses de trabajo, empezando por septiembre del 2012 hasta julio del 2013. No obstante en nigún momento ha tenido una ocupación plena, puesto que tenía otro tipo de ocupaciones incompatibles con la dedicación íntegra del proyecto.

La idea del proyecto empezó al poco de terminar el proyecto fin de carrera de la ingeniería técnica: al ver la utilidad que estaba teniendo la herramienta dentro del grupo UCASE (dirigido por la profesora Inmaculada Medina), se decidio potenciar dicha herramienta. Con la ayuda del profesor Antonio García Domínguez, se fué matizando qué necesitaba el grupo para poder sacarle el máximo partido a la herramienta. El PFC se desarrollará a la par que se cursan las últimas asignaturas de la carrera.

3.1. Etapas

La metodología usada para el modelo del software ha sido un desarrollo iterativo por prototipos [6]. En la etapa de inicialización del desarrollo iterativo, se realizó una versión completa del editor y un esbozo del concepto de estrategia de generación. En las siguientes iteraciones se fueron añadiendo y mejorando las demás funcionalidades del

sistema.

El desarrollo seguido se puede dividir en distintos apartados que se especificarán en los siguientes puntos.

3.1.1. Elicitación de requisitos

Los requisitos fueron analizándose a través de varias reuniones con los miembros del grupo de investigación, aunque fueron afinándose con posteriores reuniones con el profesor Antonio García Dominguez. Los requisitos fueron incrementándose a lo largo del proyecto, por ejemplo se añadieron más estrategias generadoras.

3.1.2. Estudio de tecnologías

En esta fase se ha llevado el estudio de varias tecnologías, gran parte prácticamente desde cero. Algunas de ellas venían impuestas por el grupo y otras se fueron eligiendo a medida que se desarrollaba. Entre las tecnologías usadas cabe destacar el lenguaje de programación Java usando de framework Eclipse, *JUnit* y *Mockito* para las pruebas unitarias, *SVN* para el control de versiones, *Xtext* para generar el AST como el editor, LaTeX para realizar la documentación y *Maven* con *Tycho* para gestión y construcción del proyecto.

3.1.3. Plugin con *Xtext*

En este apartado se estudió e implementó el editor para el lenguaje spec. Se llevó a cabo un plugin para eclipse, ya que el grupo de investigación UCASE suele usar dicho framework.

3.1.4. Plugin de ejecución

Para poder invocar al núcleo de la aplicación desde el editor, es necesario la implementación de otro plugin para eclipse que lleve a cabo dicha función.

3.1.5. Diseño de la estrategia

En este punto se estudió cómo realizar el concepto de estrategia generadora. Para ello, se tuvo que diseñar de manera muy específica un conjunto de estructuras que fuesen capaces de no sólo recoger la información, si no que también fuese mantenible y fácilmente ampliabe con más estrategias.

3.1.6. Cambio de la gramática

En este apartado se cambio la herramienta usada para la gramática, *ANTLR*, por *Xtext*. También se mejoró la gramática para que, por ejemplo admitiese lista de declaración de variables. Además había que adaptarla para que pudiese aceptar el concepto de estrategia generadora y lo que ello conlleva.

3.1.7. Cambio del analizador

Al realizarse un cambio completo de herramienta para contruir el AST por la creación del editor, tuvo que implementarse desde cero todo el analizador completo. Además de cambiar para que funcionase lo que ya existia, había que mejorarlo para que acepte el concepto de estrategia generadora.

3.1.8. Implementación de estrategias

Esta es la parte más importante del sistema, se podría decir que es la corazón del mismo. Con las estructuras de tipos, entraba en juego la generación de los datos usando la estrategia generadora adecuada. Creándose como resultado los datos que luego van a ser formateados para la salida del sistema.

3.1.9. Pruebas unitarias

Cada parte del código lleva asociada unas pruebas unitarias gracias a *JUnit* y *Mockito*, esto nos permite detectar errores y poder subsanarlos. Aparte de esto, nos permite que al incluir una nueva modificación al programa poder comprobar si sigue comportándose

de la forma deseada. Esta parte ha llevado un gran esfuerzo para que la aplicación sea sólida y consistente.

3.1.10. Documentación

La memoria del este proyecto se ha ido elaborando conforme se iba desarrollando la aplicación, pero es en los últimos meses donde se ha trabajado más esta parte.

3.1.11. Paquete de instalación y líneas de ordenes

En esta fase se llevaron a cabo los ajustes necesarios para que *TestGenerator* fuera fácil de instalar y usar desde una consola a través de líneas de órdenes.

3.2. Diagrama Gantt

Se ha elaborado un diagrama Gantt para facilitar la comprensión de la distribución de las tareas. Para elaborar dicho programa, se ha usado el software *Gantt Project*. El diagrama se muestra en las figuras 3.1 y 3.2. También podemos ver las fechas de inicio y fin de cada tarea en la tabla 3.1.

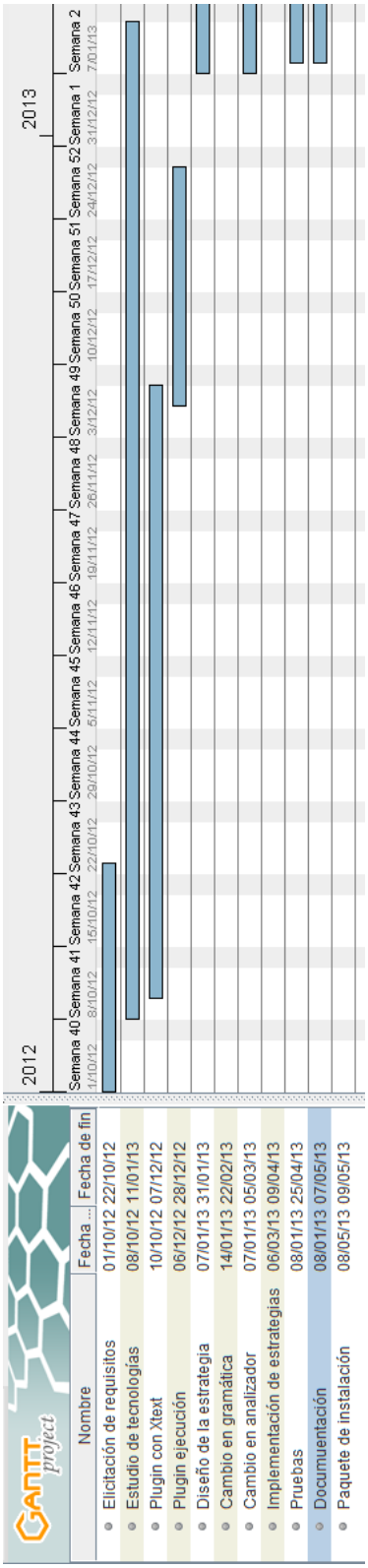

Figura 3.1.: Diagrama de Gantt I

Figura 3.2.: Diagrama de Gantt II

Fase del proceso	Fecha de inicio	Fecha de finalización	Días reales
Elicitación de requisitos	1/10/2012	22/10/2012	16
Estudio de tecnologías	8/10/2012	1/01/2013	70
Plugin con *Xtext*	10/10/2012	7/12/2012	43
Plugin de ejecución	6/12/2012	28/12/2012	17
Diseño de la estrategia	7/01/2013	31/01/2013	19
Cambio de la gramática	14/01/2013	22/02/2013	30
Cambio del analizador	7/01/2013	5/03/2013	42
Implementación de estrategias	6/03/2013	9/04/2013	25
Pruebas unitarias	8/01/2013	25/04/2013	78
Documentación	25/03/2013	10/07/2013	86
Paquete de instalación y líneas de ordenes	1/06/2013	03/06/2013	2
Total	1/10/2012	03/06/2013	159

Tabla 3.1.: Porcentajes de esfuerzo

Capítulo 4

Análisis

En este apartado hablaremos de las necesidades que debe cubrir el proyecto, es decir, los requisitos que debe cumplir. Además se realizarán los casos de uso que especifican el alcance del proyecto.

4.1. Requisitos funcionales

En este punto hablaremos de las necesidades que debe cubrir el sistema. Dichas necesidades vienen impuestas por el grupo UCASE las cuales veremos a continuación.

Los tipos de valores y restricciones a tener en cuenta en este generador serán los mismos que se implementaron en el proyecto anterior, a modo de recordatorio, el conjunto de datos que debe ser capaz de generar de forma aleatoria es el siguiente:

- Enteros.
- Números con coma flotante.
- Cadenas de caracteres.
- Fechas.
- Horas.
- Fechas y horas.

- Duraciones.
- Listas.
- N-Tuplas.

Bajo las siguientes restricciones:

- Valor máximo.
- Valor mínimo.
- Número total de dígitos.
- Número total de decimales.
- Cumplir cierta expresión regular.
- Valor válido dada una lista de valores.

Se deberá desarrollar un editor el cual facilite el trabajo de crear ficheros con la notación SPEC, dicho editor deberá constar de las herramientas típicas de edición de código como es el resaltado de sintaxis, autocompletado y detección de errores entre otras.

Por otro lado, se incluirá un concepto nuevo llamado estrategia de generación, con este concepto representaremos de qué forma vamos a generar los datos. Las distribuciones de probabilidad a implementar serán las siguientes: Gaussiana, Exponencial, Binomial y Poisson. También se implementarán distintas estrategias combinatorias. Como resultado del nuevo concepto de estrategia habrá que modificar el DSL para incluir dicho concepto, pero deberá ser un superconjunto del anterior. Esto implica modificación completa del analizador de los ficheros *spec* y generador de datos.

Para poder llevar a cabo un control de las estrategias generadoras, será necesario realizar unas estructuras o tipos de datos propios donde se guardará la información relevante de las mismas. Este sistema de tipos servirá para que la herramienta pueda comunicarse en cada una de sus fases y para poder modular la aplicación.

Todo esto seguirá siendo compatible con el actual sistema de exportación compatible con *BPELUnit*.

4.2. Estudio del mercado

Antes de empezar a desarrollar este PFC, se ha llevado a cabo un estudio de mercado para ver que herramientas existían actualmente para Java.

Básicamente se encontraron dos herramientas que se dedican a generar datos aleatorios estas son:

- *JCheck* [7]
- *QuickCheck for Java* [8]

Dichas herramientas se centran exclusivamente en dar un API para Java, de cara a ser integrados con *JUnit*. En el listado 4.1 podemos ver un ejemplo de uso de *JCheck*. El ejemplo de *QuickCheck for Java* lo veremos en el listado 4.2. En ambos ejemplos se encuentran estructura típica de un caso de prueba elaborado con *JUnit*. En el primero podemos observar dos pruebas: en la primera es una comprobación de que la suma de dos variables tipo *Money* es realizada de la forma esperada y la segunda comprueba que dado un vector a ordenar, sigue conservando el mismo tamaño. Respecto al ejemplo sobre *QuickCheck* 4.2, es una prueba que comprueba la creación de una lista ordenada eta realmente ordenada.

Se podría decir que la mayor diferencia que tiene con nuestra aplicación es el enfoque. *TestGenerator* está pensado para generar ficheros de datos genéricos a introducir en cualquier herramienta como por ejemplo *BPELUnit*, que es la que usa el grupo de investigación UCASE. Otros programas se centran exclusivamente en dar un API para Java, de cara a ser integrados en JUnit. Además, el concepto de estrategia generadora es algo novedoso por parte de este PFC y distintivo de las herramientas anteriormente mencionadas.

Aparte de esta gran diferencia de enfoque, *TestGenerator* tendrá soporte para crear fechas aleatorias, no contempladas ni por *JCheck* ni por *QuickCheck for Java*.

Listado 4.1: Ejemplo de uso de *JCheck*

```
@RunWith(org.jcheck.runners.JCheckRunner.class)
class SimpleTest {
       @Test public void simpleAdd(int i, int j) {
       Money miCHF= new Money(i, "CHF");
       Money mjCHF= new Money(j, "CHF");
       Money expected= new Money(i+j, "CHF");
       Money result= miCHF.add(mjCHF);
       assertTrue(expected.equals(result));
    }
}

@RunWith(org.jcheck.runners.JCheckRunner.class)
class SimpleTest {
       @Test public void simpleDivide(int[] somearray) {
       imply(somearray.length > 1);
       int expected= somearray.length;
       int result= MySort.sort(somearray).length;
       assertEquals(expected, result);
    }
}
```

Listado 4.2: Ejemplo de uso de *QuickCheck for Java*

```
public class SortedListTest {
	@Test public void sortedListCreation() {
		for (List<Integer> any : someLists(integers())) {
			SortedList sortedList = new SortedList(any);
			List<Integer> expected = sort(any);
			assertEquals(expected, sortedList.toList());
		}
	}

	private List<Integer> sort(List<Integer> any) {
		ArrayList<Integer> sorted = new ArrayList<Integer>(any);
		Collections.sort(sorted);
		return sorted;
	}
}
```

4.3. Requisitos de implementación

Los requisitos de implementación impuestos por el grupo UCASE son los siguientes:

- El lenguaje a utilizar para la implementación de la solución deberá ser Java [9], puesto que la mayoría de las aplicaciones del grupo están escritas en dicho lenguaje y así resultará más cómodo una posible reutilización de código para otros proyectos.
- También como requisitos indispensable, el grupo UCASE exige la realización de pruebas paramétricas mediante la herramienta JUnit [10] (§6.7).
- El grupo de investigación UCASE exige para realizar todos sus proyectos utilizar un sistema de integración continua para poder llevar un mejor seguimiento de los mismos(§6.1).

4.4. Atributos del sistema

El sistema debe tener:

- Mantenibilidad: el sistema deberá ser cómodo de mantener, dado que en un futuro podrá ser ampliado con nuevas estrategias.
- Transportabilidad: el sistema deberá tener independencia de la plataforma, para no ligar su uso a ningún sistema operativo. Aunque se recomienda el uso de alguna distribución GNU/Linux.
- Facilidad de uso: aunque este proyecto está dirigido a personas con conocimientos informáticos elevados, deberá tener facilidad de uso para que no resulte una tarea tediosa trabajar con la aplicación.
- Licencia libre: dado que se desea un uso público de la misma, la mejor manera de conseguir es otorgándole dicha licencia.

4.5. Casos de uso

En la figura 4.1 podemos ver representado el diagrama de casos de uso, usando la notación UML. En las siguientes subsecciones detallaremos los contenidos de cada uno.

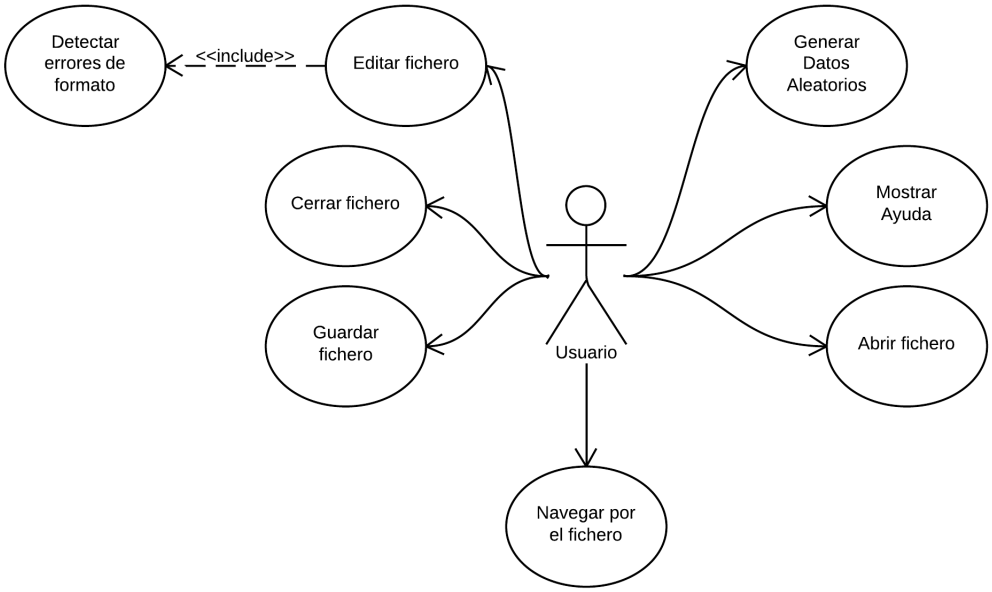

Figura 4.1.: Diagrama de casos de uso

4.5.1. Generar datos aleatorios

Este caso de uso sirve para una ejecutarlo desde línea de comando o desde el editor.

- Actor principal: Usuario que desea generar datos aleatorios.
- Precondición: Debe tener un fichero válido de entrada al programa.
- Postcondición: Genera un archivo que contendrá los datos generados aleatoriamente.
- Escenario principal:
  1. El usuario introduce como argumento la ruta de una plantilla generada por *ServiceAnalyzer*, el número de datos a generar y el formato de salida.

2. El sistema genera los datos en el formato especificado.

- Variaciones:

1a. La orden introducida no es correcta.

1. El sistema informa que la orden no es válida, muestra su forma de uso y cancela el caso de uso.

1b. El número de argumentos introducidos por la línea de ordenes es incorrecto.

1. El sistema informa que el número de argumento no es válido, muestra su forma de uso y cancela el caso de uso.

1c. El usuario no introduce el número de datos a generar.

1. El sistema utiliza el valor por defecto y continua con el caso de uso.

1d. El usuario no introduce el formato en el que desea la salida.

1. El sistema utiliza el valor por defecto y continua con el caso de uso.

4.5.2. Mostrar ayuda

- Actor principal: Usuario que desea ver la ayuda.
- Precondición: Ninguna.
- Postcondición: Desde la terminal, se visualiza la ayuda del sistema.
- Escenario principal:

1. El usuario pide al sistema que le muestre la ayuda.
2. El sistema muestra la ayuda por la pantalla.

4.5.3. Abrir fichero

- Actor principal: Usuario que desea abrir un fichero con notación SPEC.
- Precondición: Debe tener un fichero válido de entrada al programa.

- Postcondición: Abre un fichero en el editor.
- Escenario principal:
  1. El usuario elige el fichero el cual quiere abrir.
  2. El sistema abre el fichero seleccionado por el usuario.

4.5.4. Cerrar fichero

- Actor principal: Usuario que desea cerrar un fichero con notación SPEC.
- Precondición: Debe tener un fichero abierto.
- Postcondición: Cierra un fichero en el editor.
- Escenario principal:
  1. El usuario elige el fichero el cual quiere cerrar.
  2. El sistema cierra el fichero seleccionado por el usuario.

4.5.5. Guardar fichero

- Actor principal: Usuario que desea guardar o editar un fichero con notación SPEC.
- Precondición: Debe tener un fichero válido de entrada al programa.
- Postcondición: Guarda un fichero con las modificaciones realizadas por el usuario.
- Escenario principal:
  1. El usuario decide guardar unos cambios en un fichero abierto en el editor.
  2. El sistema guarda los cambios realizados por el usuario.
- Variaciones:

  2a. Es la primera vez que se guarda el fichero.

  1. El sistema pide nombre para el fichero y ubicación.
  2. El usuario introduce los datos.

2b. El fichero ya existe.

1. El sistema informa que va a sobreescribir el fichero.
2. El usuario acepta o rechaza la realización del guardado.

4.5.6. Editar fichero

- Actor principal: Usuario que desea editar un fichero con notación SPEC.
- Precondición: Debe tener un fichero válido de entrada al programa.
- Postcondición: Realiza cambios en un fichero con notación SPEC.
- Escenario principal:
  1. El usuario decide realizar unos cambios en un fichero abierto en el editor.
  2. El usuario introduce modificaciones.
  3. El sistema realiza el caso de uso “Detectar errores de formato”.

4.5.7. Detectar errores de formato

- Sistema: El sistema analiza el fichero con notación SPEC.
- Precondición: Debe de ser un fichero válido de entrada al programa.
- Postcondición: Si existe indentifica los fallos en el fichero y se los muestra al usuario.
- Escenario principal:
  1. El usuario se encuentra editando un fichero válido de entrada al programa.
  2. El sistema analiza los cambios introducidos por el usuario.
- Variaciones:

  2a. Las modificaciones introducidas no son correctas.

1. El sistema informa que las modificaciones no cumplen el formato del fichero.
2. El sistema vuelve al primer paso del caso de uso.

4.5.8. Navegar por el fichero

- Actor principal: Usuario que desea navegar por un fichero con notación SPEC.
- Precondición: Debe tener un fichero abierto.
- Postcondición: Cierra un fichero en el editor.
- Escenario principal:
  1. El usuario elige a que parte del fichero desea dirigir el cursor.
  2. El sistema traslada el cursor a dicho punto.

Capítulo 5

Diseño

En este capítulo hablaremos de las condiciones que nos llevarán a tomar las decisiones de diseño, así como que decisiones se ha llevado a cabo y de la estructura que tendrá el proyecto.

5.1. Arquitectura general del sistema

En este punto, se dará a conocer los aspectos relativos de cómo será la estructura de este sistema.

En este proyecto, se ha seguido el patrón arquitectónico «Pipes and Filters» (tuberías y filtros) [11]. Dicho patrón arquitectónico proporciona una estructura para sistemas que procesan un flujo de datos. Cada paso de procesamiento se encapsula en un componente de filtro, los datos se pasan a través de tuberías entre los filtros adyacentes. La recombinación de filtros, permite crear sistemas relacionados.

Este proyecto, se encuentra dividido en tres grandes bloques (filtros). Esta división está basada en la funcionalidad que cumplen cada uno de los bloques. Los bloques son:

- Analizador: encargado de analizar los ficheros de entrada y montar el sistema de tipos necesario para poder trabajar con ellos.
- Generador: bloque principal del sistema encargado de la generación de la forma

adecuada de los datos.

- Formateador: encargado de prototipar la salida de los datos generados en la fase anterior, es decir, darle el formato adecuado.

La división en estos tres bloques, nos aporta grandes ventajas. Entre ellas podemos destacar:

- Desarrollo: con esta estructura, es más sencillo y cómodo usar la metodología llevada en este proyecto que ha sido iterativa por prototipo.
- Corrección de errores: al estar claramente diferenciadas las funcionalidades, es más cómodo identificar problemas y corregir errores, asegurando que el resto de partes funcionen correctamente.
- Ampliación del sistema: si en algún momento necesitamos ampliar algún formateado extra o ampliar los tipos de ficheros de entrada al programa, gracias a dicha división es menos costoso y reduce la aparición de errores.
- Reutilización de código: puesto que el proyecto pertenece al grupo de investigación UCASE, es más sencillo la reutilización de alguna de sus partes en otros proyecto, cosa que actualmente se lleva a cabo.

En la figura 5.1 podemos ver cómo se comunican los bloques entre sí.

Para que quede más clara la visión global del sistema, se aporta una imagen que identifica cada parte dentro del sistema en la figura 5.2.

5.2. Sistema de tipos de TestGenerator

El dominio de datos del sistema deberá ser equivalente al conjunto de datos y restricciones que puede tomar del catálogo generado por *ServiceAnalyzer*.

Dicho conjunto de datos es el siguiente:

- `string`: Cadena de caracteres válidos Unicode e ISO/IEC 10646.

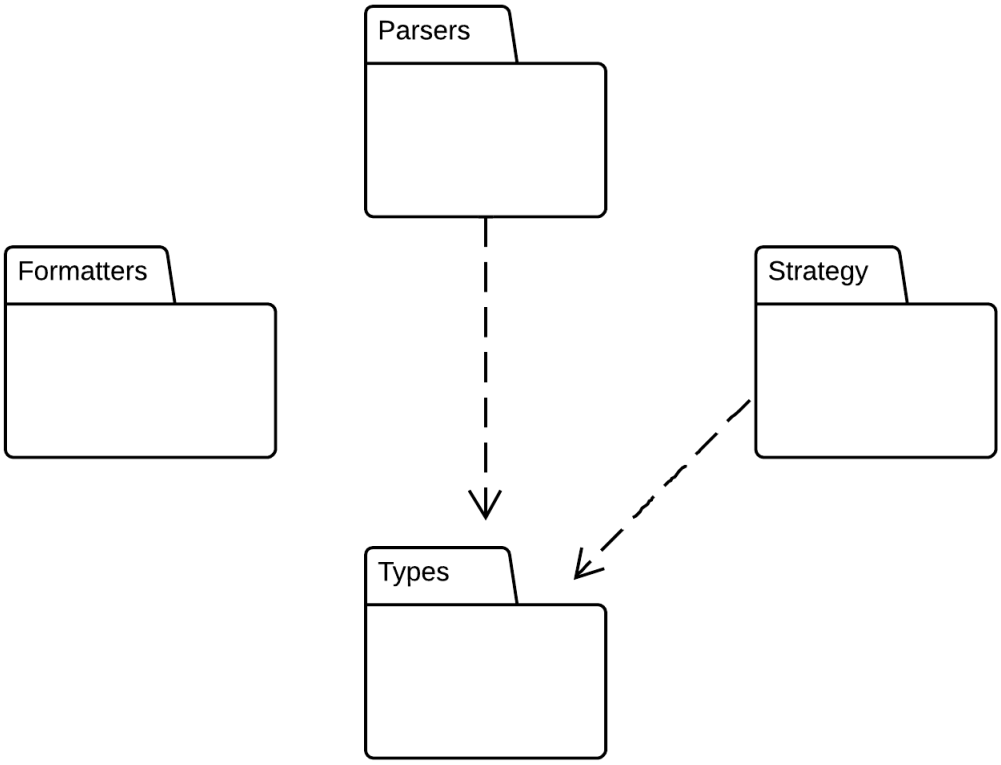

Figura 5.1.: Diagrama de los paquetes

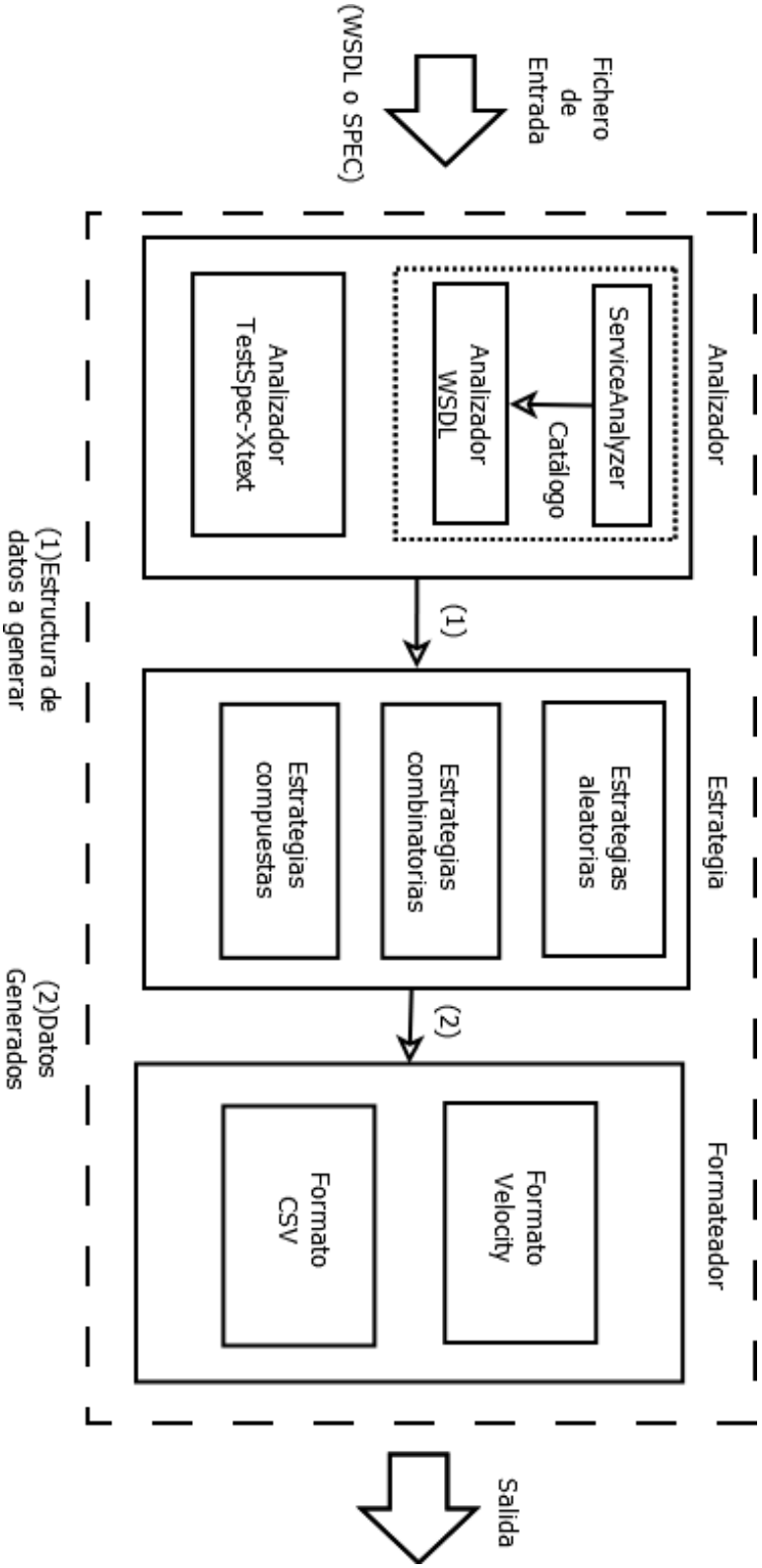

Figura 5.2.: Estructura del sistema

- `int`: Enteros que pueden almacenarse en 32 bits.
- `float`: Representa números de coma flotante de 32 bits, según el estándar IEEE 754.
- `date`: Define un día concreto del calendario Gregoriano con el formato `YYYY-MM-DD`, como por ejemplo, “2003-10-21”.
- `time`: Define una hora concreta con el formato `hh:mm:ss`, como por ejemplo, “10:21:23”. El número de segundos puede incluir dígitos decimales de precisión arbitraria si se desea. Las horas se datan según el sistema de 24 horas.
- `dateTime`: Define un instante de tiempo concreto, usando el calendario Gregoriano. El formato es `YYYY-MM-DDThh:mm:ss`, por ejemplo, “2003-10-21T20:30:13”. La T separa la fecha de la hora. Se puede añadir también una Z opcional, y + o - hh:mm al final para indicar una zona horaria diferente. Usamos Z si la hora se ajusta GMT (Greenwich Mean Time) o o a UTC (Coordinate Universal Time), o utilizaremos las horas y minutos adicionales para indicar diferencias respecto al GMT.
- `duration`: Expresa una duración en un espacio de 6 dimensiones. El formato es `PnYnMnDTnHnMnS`, donde `nY` representa el número de años, `nM` el número de meses, `nD` el número de días `nH` el número de horas, `nM` el número de minutos y `nS` el de segundos. `P` es un indicador obligatorio que debe estar el primero, mientras que T es el carácter que separa la fecha de la hora y únicamente debe aparecer si se indica una hora. Ninguno de los elementos es obligatorio ni tiene limitación de rango. También se pueden indicar duraciones negativas precediéndolas del signo ’-’ (si se omite el signo, se tratarán como duraciones positivas). Un valor de tipo duration podrá ser, por ejemplo, “P1DT2S” (duración de un día y dos segundos).

También se van a contemplar algunos tipos contenedores, en concreto las listas y las tuplas:

- `list`: Una lista es una colección de elementos que contiene el mismo tipo de dato.

- `tuple`: Una tupla es una colección de elementos los cuales pueden ser de distintos tipos.

El conjunto de restricciones que pueden aparecer en la plantilla o ficheros *spec* es el siguiente:

- `element`: Esta restricción es obligatoria si el tipo es alguno de los contenedores, representa el tipo de elemento que va a contener. En el caso de las tuplas podríamos necesitar una lista de tipos, por lo que el valor de `element` es una lista ordenada de cadenas separadas por coma.

- `min`: Este atributo tiene un significado u otro en función del tipo al que se aplique. Aplicado a tipos numéricos, representa el límite inferior inclusivo del espacio de valores del tipo. En el caso de que se aplique a un tipo cadena, indica la longitud mínima que ésta ha de tener. Sin embargo, si es un tipo `list`, `min` indica el número mínimo de elementos que puede contener la lista.

- `max`: Análogamente a la definición de `min` se le puede aplicar sustituyendo la palabra mínimo por máximo.

- `values`: Restringe el espacio de valores de un determinado tipo al conjunto de valores especificados.

- `pattern`: Restringe el espacio de valores de un determinado tipo, restringiendo el espacio léxico a literales que siguen un determinado patrón. El valor debe ser una expresión regular.

- `fractionDigits`: Controla el número de decimales que deberá contener `float`.

- `totalDigits`: Controla el número total de dígitos que tendrá un número.

En las figuras 5.3 y 5.4 podemos ver el diagrama de clases que representa cómo vamos a guardar los tipos y sus restricciones.

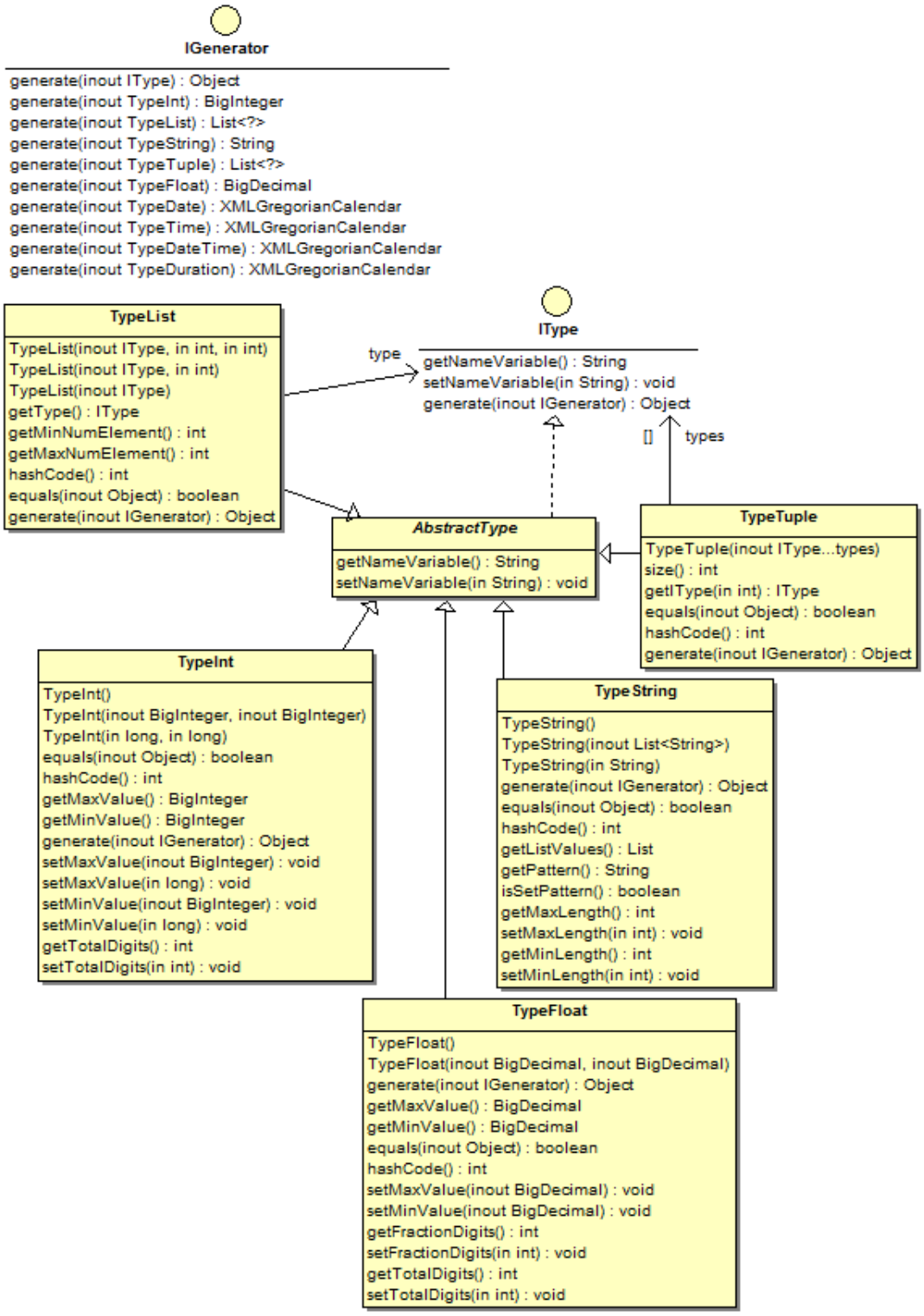

Figura 5.3.: Diagrama de los tipos I

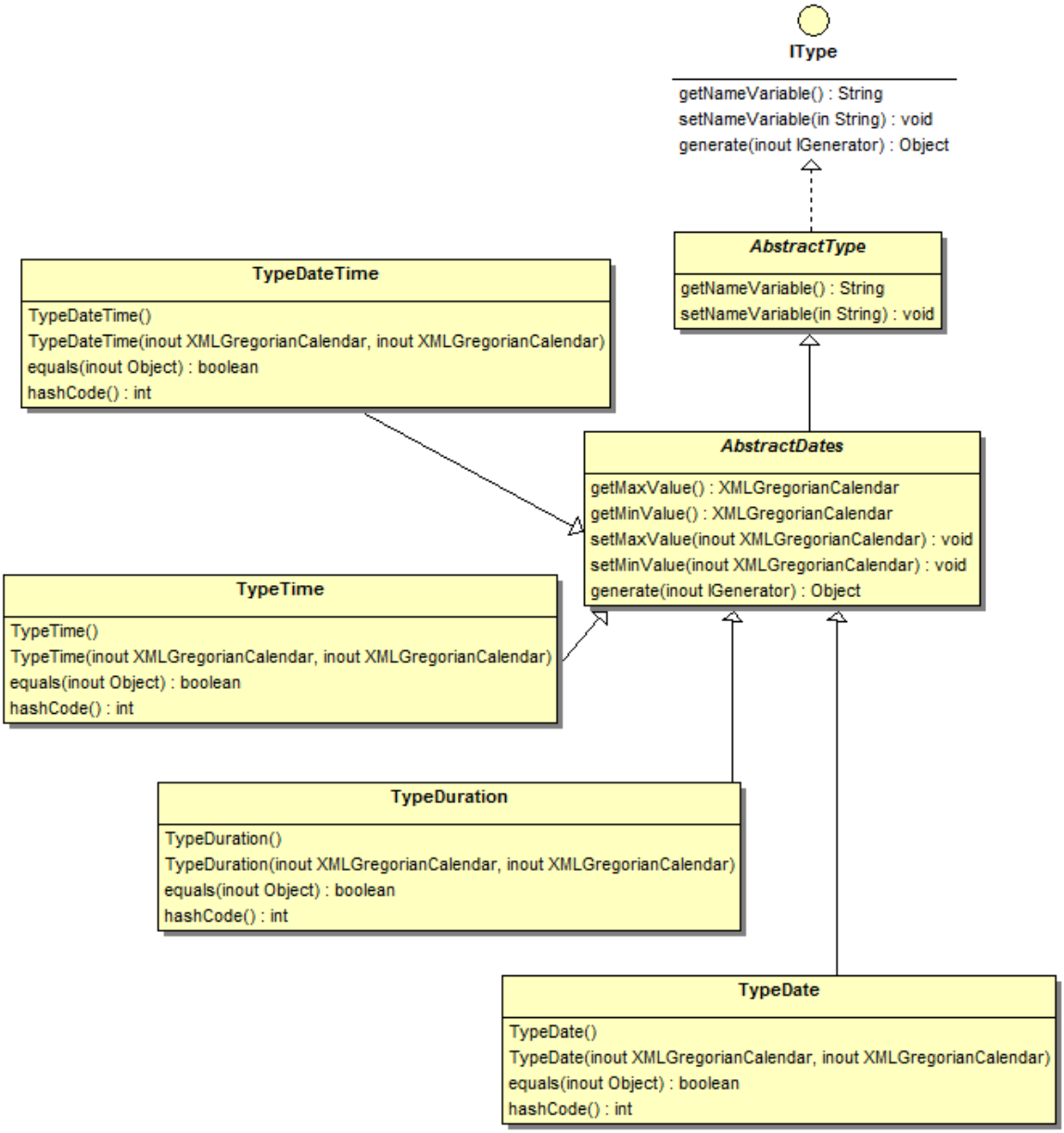

Figura 5.4.: Diagrama de los tipos II

5.3. Analizador de catálogos de ServiceAnalyzer

Este bloque estructural del proyecto, es el encargado de analizar los ficheros que recibe de entrada con el fin de crear una estructura de datos que guarde la información relevante para, en una etapa más tardía, poder generar los datos aleatorios. El sistema es capaz de analizar los catálogos generados por *ServiceAnalyzer* a partir de un fichero *wsdl* del grupo de investigación.

Para el análisis de los ficheros *wsdl* utilizaremos *ServiceAnalyzer*. Dicho programa nos generará un catálogo que será el que analizaremos utilizando las clases del propio *ServiceAnalyzer*, las cuales utilizan la herramienta *XMLBeans*. Gracias a dicha herramienta seremos capaz de recorrerlo, y a partir de esto, recoger la información en unas estructuras creadas para tal fin que contendrá las restricciones de los datos que debemos generar.

5.4. Analizador de especificaciones TestSpec

5.4.1. Motivación

Aunque los resultados con los catálogos de *ServiceAnalyzer* son prometedores, nos damos cuenta que la herramienta queda muy limitada y acotada ya que la utilidad de la herramienta queda restringida a dichos catálogos. Por todo ello llegamos a la conclusión de que es necesario pensar en alguna manera de sacarle el mayor provecho posible a toda la infraestructura creada para la misma. Así surge la idea de realizar un lenguaje de dominio específico con el cual poder exprimir al máximo todas las oportunidades que este proyecto nos sugiere y nace el lenguaje TestSpec: un lenguaje capaz de representar nuestra estructura de datos con sus restricciones para generar casos de prueba baja un conjunto de estrategias personalizables.

5.4.2. Sintaxis abstracta

Una especificación TestSpec estará dividida en tres bloques: en el primero aparecerán las definiciones de los tipos con sus restricciones la cual denominaremos `typedef`, en el

segundo aparecerán las declaraciones de los tipos el cual llamaremos `declaration` y tercero y último un bloque opcional que denominaremos `strategy` en el cual especificamos cómo se van a generar los datos. Si dicho bloque no aparece o en él no hace mención a algún dato definido se generará usando una distribución uniforme.

Las palabras reservadas para especificar las restricciones son las siguientes:

- min
- max
- values
- totalDigits
- fractionDigits
- element
- pattern

Una vez que tenemos la especificación, necesitamos construir nuestro metamodelo para representar los conceptos necesarios de nuestra gramática. En la figura 5.5 podemos ver un diagrama con el metamodelo de nuestro DSL. En ella podemos apreciar el concepto `Model` que sirve para empezar a generar nuestro modelo del cual parten los conceptos `Entry` o `Strategy`. Del concepto `Entry` cuelgan las definiciones de los tipos (`Typedef`) y la declaraciones de las variables(`VariableList`). A partir de estas estructuras cuelgan toda la información necesaria para representar nuestros tres grandes bloques: definiciones, declaraciones y estrategias.

5.4.3. Sintaxis concreta

Para representar nuestro DSL, necesitamos realizar una gramática capaz de representar toda nuestra sintaxis. Dicha gramática la podemos observar en el listado 5.1.

Listado 5.1: Gramática del lenguaje TestSpec

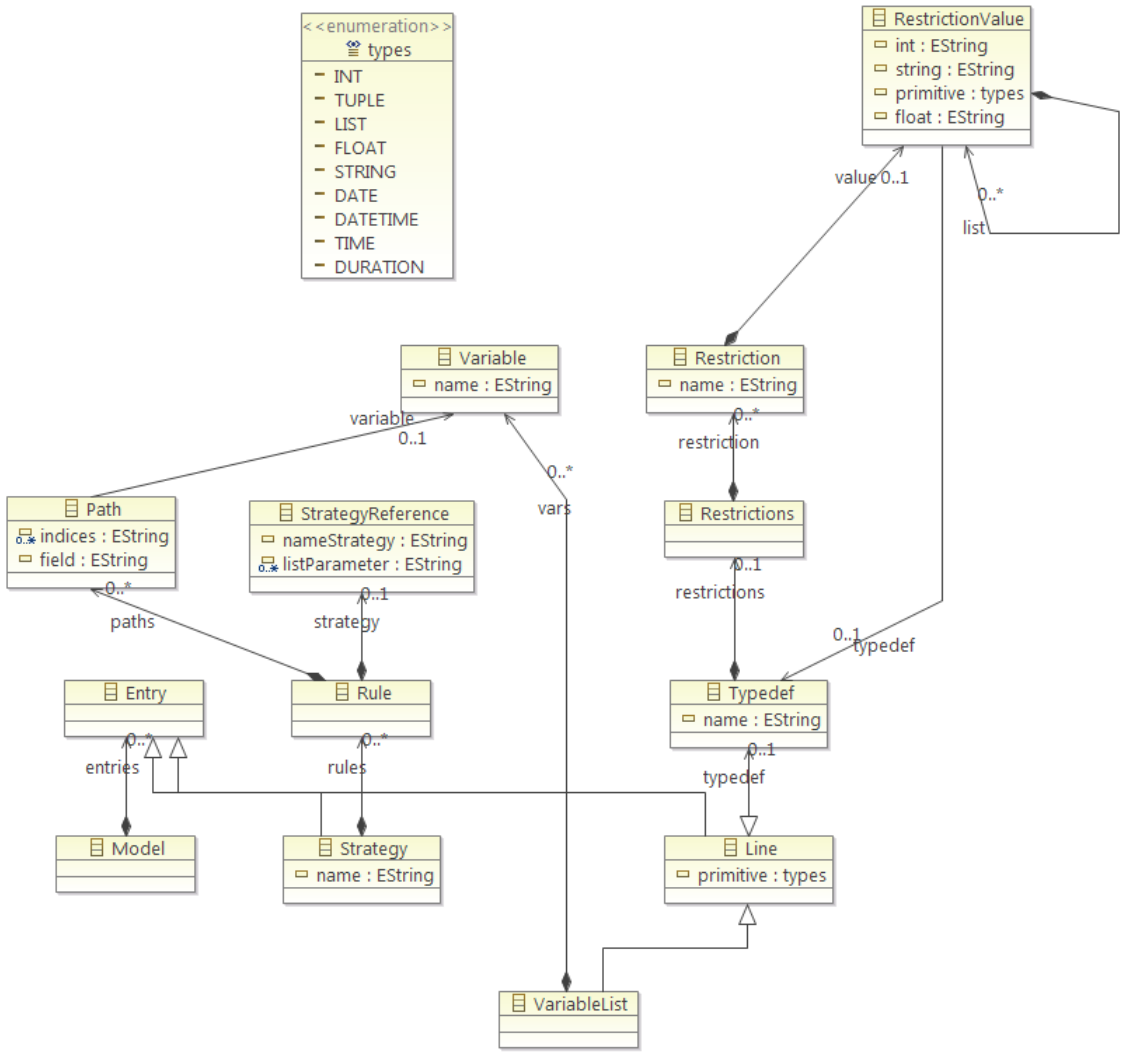

Figura 5.5.: Metamodelo de TestSpec

```
Model: Entry*;

Entry: Line ';' | Strategy;

Line: Typedef | VariableList ;

Typedef: 'typedef' (types | Typedef) Restrictions? ID;

VariableList: (types | Typedef) Variable (',' Variable)*;

Variable: ID;

Strategy: 'strategy' ID '{' (Rule)* '}';

Rule: Path(',' Path)* ':' StrategyReference;

Path: Variable ('[' INTEGER ']')* ('.' 'size')?;

StrategyReference: ID '(' StrategyParameter (',' StrategyParameter)*)? ')';

StrategyParameter: INTEGER | FLOAT | STRING;

types
    : INT = "int"
    | TUPLE = "tuple"
    | LIST = "list"
    | FLOAT = "float"
    | STRING = "string"
    | DATE = "date"
    | DATETIME = "dateTime"
    | TIME = "time"
    | DURATION = "duration"
    ;
```

```
Restrictions: '('{Restrictions} (Restriction (',' Restriction)* )? ')';

Restriction: ID '=' RestrictionValue ;

RestrictionValue: INTEGER | STRING
       | types | Typedef | FLOAT
       | {RestrictionValue}'{' (RestrictionValue (',' RestrictionValue)* )? '}'
       ;

INTEGER
       : ('+'|'-')?('0'..'9')+ ;

FLOAT
    :   ('+'|'-')?('0'..'9')+ '.' ('0'..'9')* EXPONENT?
    |   '.' ('0'..'9')+ EXPONENT?
    |   ('+'|'-')?('0'..'9')+ EXPONENT
    ;

EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;

STRING :
  '"' ( '\\'('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\'|'0')
  | !('\\'|'"') )* '"' |
  "'" ( '\\'('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\'|'0')
  | !('\\'|"'") )* "'";
```

5.4.4. Ejemplos

Para que quede más claro los ejemplos, vamos a recordar cómo se estructura una especificación TestSpec. Una especificación TestSpec estará dividida en tres bloques: en el primero aparecerán las definiciones de los tipos con sus restricciones la cual denominare-

Listado 5.2: Ejemplo de especificación TestSpec

```
1 typedef int (min=1, max=6) CaraDado;
2 CaraDado dado;
3
4 strategy dadoGaussian{
5        dado : Gaussian(0,1)
6 }
```

mos `typedef`, en el segundo aparecerán las declaraciones de los tipos el cual llamaremos `declaration` y tercero y último un bloque opcional que denominaremos `strategy` en el cual especificamos cómo se van a generar los datos. Si dicho bloque no aparece o en él no hace mención a algún dato definido se generará usando una distribución uniforme. Veamos un ejemplo.

En el listado 5.2 podemos observar que las líneas que pertenecen al bloque de definiciones aparece la palabra reservada `typedef`, a continuación el tipo que vamos a definir (`int`) y después, aparecen el conjunto de restricciones formadas por los pares clave-valor (`min=1,max=6`). Para terminar, al final de la línea aparece el nombre que le vamos a dar a dicha definición (`CaraDado`) precedido del carácter fin de línea “`;`”.

A su vez un `typedef` puede servir de tipo para otro `typedef` posteriormente declarado.

En las sentencias de las declaraciones, aparecerá el nombre de una de las definiciones anteriores (`CaraDado`) y el identificador que le vamos a otorgar a la variable (`dado`) precedido del carácter fin de línea “`;`”.

Por último aparecen las estrategias, en este caso solo una (`dadoGaussian`) que indica que la forma de generar los datos es usando una distribución gaussiana. Podemos ver que este bloque comienza por la palabra reservada `strategy` seguido del nombre que le vamos a dar a la estrategia. Luego dentro tenemos un bloque(definido por `{}`) donde aparecerá la forma que queremos generar los datos. Para ello, ponemos la lista de variables que queremos generar, el caracter reservado `:` y la estrategia que queremos usar.

Este tipo de estructura hace de TestSpec un lenguaje muy cómodo y sencillo de usar para nuestro objeto: especificar un conjunto de restricciones asociadas a un tipo de dato.

Mostraremos un ejemplo algo más elaborado para que quede clara su potencia en el listado 5.3. En el podemos apreciar la definición de seis tipos:

- Boolean: tipo string que admite los valores “true” o “false”.
- CaradaDado: entero que admite los valores del rango [1,6].
- ResultadoTirada: una tupla compuesta por un string y CaraDado.
- ListaNoVaciaInt: definimos una lista de enteros que mínimo contendra un entero.
- PositiveNumber: un entero que como valor más bajo será el 0.
- SmallNumber; entero comprendido en el rango [0,100].

Luego podemos ver la definición de las variables y por último vemos la estrategia, en la que apreciamos que queremos generar la variable “tirada” siguiendo una distribución gaussiana con media igual a cero y distribución estandar uno. Luego aparece que deseamos generar las variables “myBoolean” y “my‘Number” siguiendo el algoritmo Comb.

5.5. Estrategias básicas

5.5.1. Estrategias aleatorias

Las distintas estrategias que podemos usar son las que vamos a definir a continuación y un ejemplo de su sintaxis lo podemos ver en el listado 5.4.

- Uniforme: La estrategia de generación de que sigue una distribución uniforme es la más comúnmente utilizada en la generación de casos de pruebas. Dicha estrategía es la usada por defecto cuando no se especifica que usar, también esta estrategia era la única existente antes de la mejora del proyecto.

Listado 5.3: Ejemplo complejo de especificación TestSpec

```
typedef string (values={"true", "false"}) Boolean;
typedef int (min=1, max=6) CaraDado;
typedef tuple (element = {string, CaraDado}) ResultadoTirada;
typedef list (min=1, element = int) ListaNoVaciaInt;
typedef int (min=0) PositiveNumber;
typedef PositiveNumber (max=100) SmallNumber;

Boolean myBoolean;
ResultadoTirada tirada;
ListaNoVaciaInt numeros;
SmallNumber myNumber;

strategy Estrategia{
       tirada: Gaussian(0,1)
       myBoolean, myNumber: Comb()
}
```

- Gaussiana: Esta estrategia de generación genera datos siguiendo una distribución normal o gaussiana. Dicha estrategia recibe la media y la desviación típica (por defecto la media es 0 y la desviación tipica 1). La importancia de esta distribución radica en que permite modelar numerosos fenómenos naturales, sociales y psicológicos ya que en el mundo real la mayoría de los datos siguen dicha distribución. Un ejemplo de cómo usar dicha distribución lo podemos ver en el listado 5.4.

- Poisson: La estrategia de distribución de Poisson recibe como parámetro la media, sienda esta por defecto 1, y esta estrategia representa, a partir de una frecuencia de ocurrencia media, la probabilidad que ocurra un determinado número de eventos durante cierto periodo de tiempo. Fue descubierta por Siméon-Denis Poisson, que la dio a conocer en 1838 en su investigación sobre la probabilidad de los juicios en materias criminales y civiles.

- Exponencial: Mientras que la distribución de Poisson describe las llegadas por unidad de tiempo, la distribución exponencial estudia el tiempo entre cada una de estas llegadas. Si las llegadas son de Poisson el tiempo entre estas llegadas es exponencial. Mientras que la distribución de Poisson es discreta la distribución exponencial es continua porque el tiempo entre llegadas no tiene que ser un número entero. Esta distribución se utiliza mucho para describir el tiempo entre eventos. Más específicamente la variable aleatoria que representa al tiempo necesario para servir a la llegada. Ejemplos típicos de esta situación son el tiempo que un medico dedica a una exploración, el tiempo de servir una medicina en una farmacia, o el tiempo de atender a una urgencia.

5.5.2. Estrategias combinatorias

- Sum: Este es el algoritmo combinatorio más simple de los implementados. Dicho algoritmolo que hace es que dado un número fijado que deba dar como resultado la suma, genera de forma aleatoria distintos valores que cumplen dicho resultado.

Listado 5.4: Ejemplo de sintaxis de estrategias

```
int a;
strategy distGaussiana{
 a:Gaussian(0,1)
}
strategy distPoisson{
 a:Poisson(2)
}
strategy distExponencial{
 a:Exponential(20)
}
```

Por ejemplo: si el resultado de la suma debe de ser 4 y tenemos dos varibales, las posibles soluciones son las combinaciones [0,4], [1,3], [2,2], [3,1] y [4,0].

- EachChoice: Este algoritmo fue propuesto por Ammann y Offutt. La idea básica detrás de la estrategia de combinación de cada opción es incluir a cada valor de cada parámetro en al menos un caso de prueba. Esto se consigue mediante la generación de casos de prueba mediante la selección de valores no utilizados sucesivamente para cada parámetro. Por lo tanto, el algoritmo es muy simple: mientras que hay valores no visitados, una nueva combinación se construye. Si un conjunto tiene cualquier valor no utilizado, éste se inserta en la combinación, de lo contrario, se introduce un valor aleatorio. Este algoritmo produce pequeñas series de pruebas, pero la cobertura de caja blanca alcanzada es muy buena. En la web [12] podemos probar este tipo de algoritmo y ver como se comporta.

- Comb: El algotitmo Comb (en español llamado Peine) se basa en generar primero todas las posibles combinaciones existentes y enumerarlas. A continuación, realiza la selección de las combinaciones más distintas existentes en función de la forma que han sido enumeradas. Por lo tanto, en una primera iteración, el algoritmo agrega la primera, la última y la posición central: suponiendo que existen tres

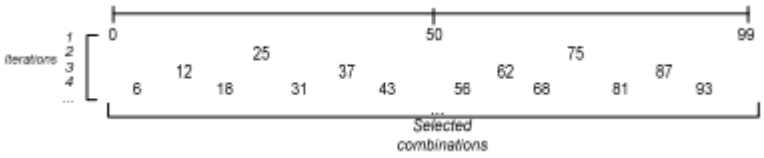

Figura 5.6.: Ejemplo visual del algoritmo Comb

conjuntos con 4, 5 y 5 elementos, habrá un máximo de 4 * 5 * 5 = 100 combinaciones: por lo tanto, en la primera iteración se incluirán las combinaciones en las posiciones 0, 99 y 50. A continuación, en la segunda iteración, los elementos colocados en el medio de los intervalos [0, 50] y [50, 99] se incluyen. Esto produce la adición de las combinaciones 25 y 75. En las iteraciones restantes, se añaden los elementos intermedios de los intervalos correspondientes a la iteración anterior. El algoritmo se detiene cuando se ha alcanzado un número prefijado de combinaciones. En la imagen 5.6 tenemos una interpretación visual de cómo elige las combinaciones y podemos apreciar que tiene forma de peine, de ahí su nombre. Para más información podemos visitar el siguiente artículo [13].

- EqualValue: esta estrategia se basa en proporcionar el mismo valor a un cojunto de variables. La forma en que se generará dicho valor puede ser cualquiera de las estrategias existentes.

5.6. Estrategias compuestas

Al tener estrategias combinatorias (estrategias que los valores a generar dependen de las variables entre sí) nos aportan una gran potencia al generar casos de pruebas pero, hay que tener especial cuidado con el tipo complejo lista. Veamoslo con un ejemplo: En el listado 5.5 podemos ver que hemos definido un tipo int llamada EnteroPositivo y un tipo list de EnteroPositivo, además hemos definido una estrategia con la cual decimos que utilice el algoritmo Comb para generar los elementos de la lista con la variable “e”. Es decir estamos diciendo que cada elemento de la lista use el algoritmo Comb con una

Listado 5.5: Ejemplo de necesidad de ámbitos

```
typedef int (min=0, max=99) EnteroPositivo;
typedef list (elements={EnteroPositivo}) Lista;

EnteroPositivo e;
Lista l;

strategy Estrategia {
    l[1], e: Comb()
}
```

variable que esta fuera de ella y esto no tiene sentido ya que el número de apariciones dentro de la lista es indeterminada y la de los elementos "e" esta definido. Otro ejemplo podría ser si usáramos una estrategia que otorgase el mismo valor a dos variables y una de ellas se encuentra dentro de la lista, tendríamos una lista que todos los valores serían iguales y esto empobrece nuestro casos de prueba. Por todo ello surge la necesidad de incluir el concepto de ámbito para paliar este tipo de situaciones.

Un ámbito es una zona en la cual los tipos que pertenecen al mismo se pueden combinar para generar datos entre sí. Los ámbitos quedarán estructurados de la siguiente manera: los datos simples y las tuplas estarán en el mismo ámbito y cada elemento anidado de una lista estará en otro ámbito.

Para implementar dicha lógica utilizaremos dos clases: la clase *Scope*, que representa un ámbito y la clase *ScopeRule* que representa la regla de generación.

Los atributos de la clase *ScopeRule* son los siguiente:

- types: son los tipos implicados en la generación.
- strategy: representa la estrategia con la cual se generarán los datos.

Y los de la clase *Scope* son los siguiente:

- types: son los elementos raiz del ámbito.

Listado 5.6: Ejemplo de ámbitos

```
typedef tupla (elements={float, int}) tupla;
typedef list (elements={tupla}) listaTupla;

listaTupla l;

strategy distGaussian {
    l[1][1], l[1][2]: Gaussian(3, 1)
}
```

Listado 5.7: Ejemplo de tipos con ámbitos

```
types = {l} //raíz del árbol
assigments = null//primer ámbito sin regla
nestedScopes = {tupla, {TypeFloat{Rule:TypeFloat,TypeInt
                                    -> GaussianStrategy(3,1) },
                        TypeInt{Rule:TypeFloat,TypeInt
                                    -> GaussianStrategy(3,1)}}
                            }
```

- assigments: representa la asignación de las reglas
- nestedScopes: representa los ámbitos anidados y es una relación recursa de la clase.

Lo explicaremos mejor con el ejemplo 5.6. En este ejemplo lo que queremos generar es una lista de tuplas donde los elementos float e int de la tupla se deben generar siguiendo una distribución Gaussiana. Viendo el ejemplo vemos que tenemos una lista de tupla, es una de las estructuras complejas que podemos encontrar y, sobre ella, veremos cómo se anidan los ámbitos. Los ámbitos quedarías de la siguiente forma en la clase *Scope*:

Podemos ver el primer ámbito que sería la lista y sólo podriamos tener una regla para

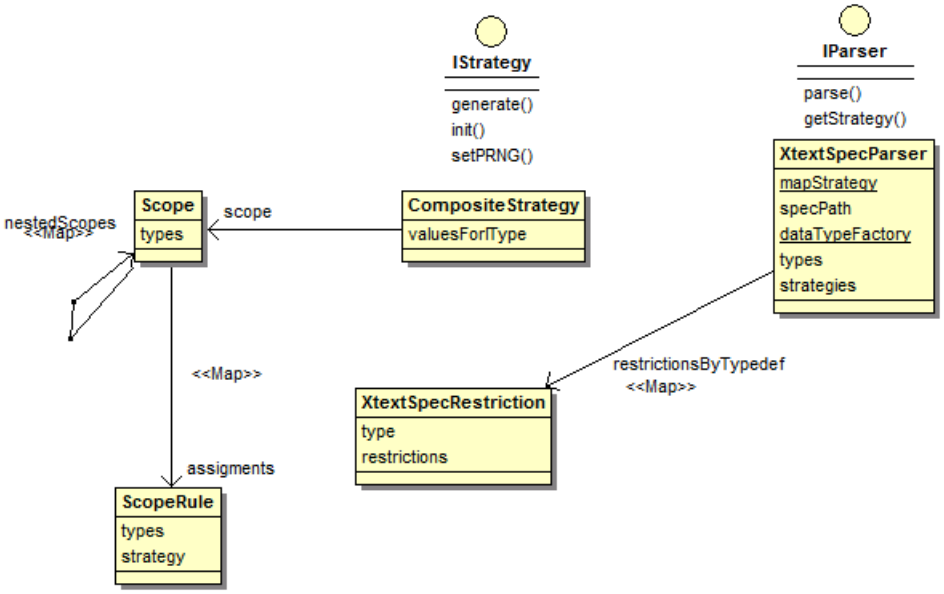

Figura 5.7.: Diagrama del analizador

definir el tamaño de la misma y ya de forma anidada tenemos de segundo ámbito, en este tenemos las reglas para definir la estrategia de generación de los tipos *float* e *int* dentro de la tupla.

En la figura 5.7 podemos observar el diagrama de clases de los analizadores.

5.7. Formateadores

Es el encargado de formatear los datos generados en el bloque anterior a un formato especifico. Los formatos a elegir pueden ser CSV o Apache Velocity.

El formato CSV (del inglés «comma-separated values») representa los datos en forma de tabla donde las columnas se encuentran separadas por comas ',' y las filas por salto de línea. En nuestro caso, la primera tupla va a representar el nombre de las variables y cada tupla posterior representará un caso de prueba. Aunque este formato es muy cómodo de usar es relativamente sencillo, ya que no permite representar por ejemplo listas o tuplas. Podemos ver un ejemplo de cómo quedaría una salida de nuestro programa en el

Listado 5.8: Ejemplo de fichero de datos CSV generado por TestGenerator

```
as_reply, accepted, ap_limit, as_limit, ap_reply, req_amount
"silent", "true", 62309, 2234, "silent", 178305
"low", "true", 103284, 2084, "true", 176109
"high", "true", 85487, 2831, "true", 63107
"high", "false", 104922, 2719, "smart", 124955
"high", "true", 16135, 2943, "true", 150697
"smart", "false", 33981, 1788, "true", 40718
"smart", "true", 67753, 1345, "silent", 145627
"smart", "true", 110294, 3692, "false", 167525
"low", "true", 11569, 3661, "smart", 198891
"silent", "false", 110002, 2223, "false", 87784
```

Listado 5.9: Ejemplo de fichero de datos Apache Velocity generado por TestGenerator

```
#set($as_reply = ["silent", "low", "silent", "low", "high"])
#set($accepted = ["true", "true", "true", "false", "false"])
#set($ap_limit = [91672, 88086, 90953, 111824, 41495])
#set($as_limit = [3496, 1371, 468, 3286, 526])
#set($ap_reply = ["true", "false", "true", "true", "false"])
#set($req_amount = [122740, 114966, 151674, 73523, 161326])
```

listado 5.8.

El formato Apache Velocity es un motor de plantilla basado en Java, propone un formato donde se verá una variable y el conjunto de valores que va a tomar. Lo explicaremos desde el ejemplo

La estructura sigue el patrón `#set($NombreVariable = [valor1, valor2])`. Cada valor será un caso de prueba, si queremos incluir una lista como caso de prueba, sus elementos vendrán entre corchetes '[' ']' reservando la pareja de corchetes más general para especificar los valores.

En la figura 5.8 podemos observar el diagrama de clases de los formateadores, po-

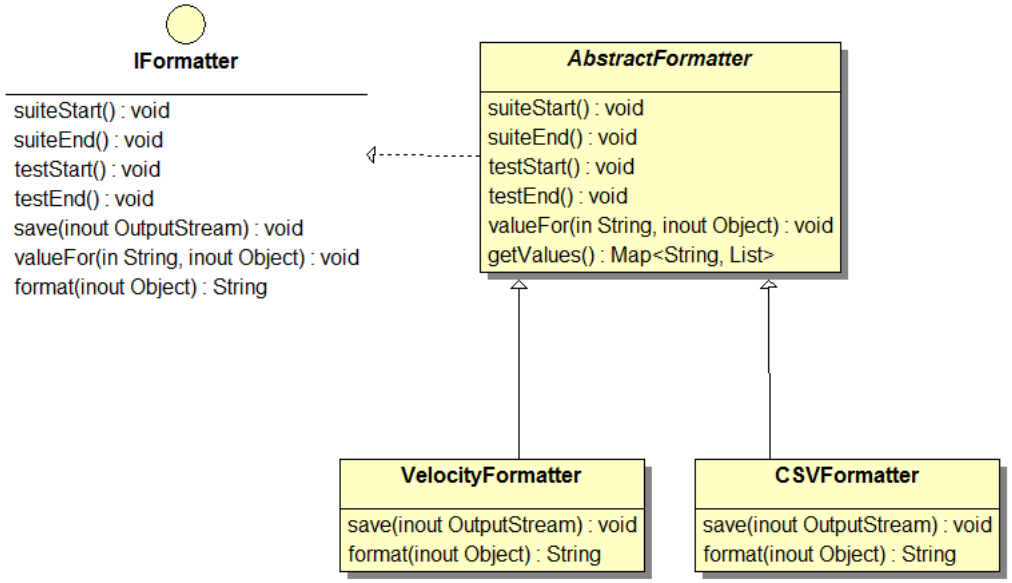

Figura 5.8.: Diagrama del formateador

demos ver que aparecen los métodos *suiteStart* y *suiteEnd* que sirve para englobar el conjunto de casos de prueba, con los métodos *testStart* y *testEnd* cubrimos una prueba concreta y con el método *valueFor* añadimos un valor para un tipo concreto.

Capítulo 6

Implementación y pruebas

En este punto hablaremos de los aspectos más técnicos, cuáles han sido las herramientas y tecnologías que se han elegido para el desarrollo de la solución. También hablaremos de las pruebas que se han llevado a cabo.

Este proyecto se ha utilizado el paradigma de programación orientado a objetos, en particular se ha utilizado el lenguaje Java. La elección del lenguaje viene como un requisito del grupo para poder reutilizar código existente (*ServiceAnalyzer*) como también pueda ser este código reutilizado en proyectos futuros.

Las grandes ventajas que nos aporta el paradigma de programación orientado a objeto [14] son las siguiente:

- Cercanía de sus conceptos a los del mundo real.
- Proceso de desarrollo más sencillo y más rápido.
- Facilita reutilización de diseño y códigos.
- Modificaciones, extensiones y adaptaciones más sencillas
- Sistemas más estables y robustos.

6.1. Integración continua

Como ya hemos mencionado anteriormente, el grupo UCASE trabaja con un modelo de integración continua.

El esquema básico montado en el grupo, consta de la interacción de cinco herramientas de distribución libre: *Jenkins*, *Subversion*, *Maven*, *Nexus* y *Sonar*.

6.1.1. Jenkins

Jenkins es una versión de la herramienta *Hudson* creada por la comunidad de software libre tras disputas con Oracle acerca del control de la marca registrada y la infraestructura del proyecto. *Hudson* es una herramienta de integración continua de código abierto, creada por Kôsuke Kawaguchi (antiguo empleado de Sun) en su tiempo libre, que se encarga de monitorear la ejecución de tareas repetidas, tales como la creación de un proyecto o la ejecución de tareas automáticas. Sus principales características son:

- Es fácil de instalar, ya que se distribuye como un fichero war.
- Todas las tareas de administración se realizan usando una interfaz web, lo cual facilita enormemente su configuración.
- Soporta notificación vía IM (Instant Messaging), e-mail y RSS (Really Simple Syndication).
- Genera gran cantidad de informes para JUnit y TestNG.
- La herramienta puede extenderse y personalizarse fácilmente mediante «plugins».
- Soporta CVS y Subversion para el control de cambios.

6.1.2. Subversion

Subversion es un sistema que permite almacenar todas las versiones de un árbol de ficheros, pudiendo así manipular todas las revisiones de cualquier fichero en cualquier

momento. Además de servir como una medida de seguridad contra la pérdida de información accidental, agiliza los cambios drásticos, ya que no hay que establecer medidas especiales por si fallaran: se pueden revertir los cambios hechos en cualquier momento.

Subversion es un sistema de control de versiones creado para reemplazar a *CVS*. Las ventajas que nos aporta en comparación con *CVS* son:

- Puede mantener revisiones de directorios completos.
- Establecer propiedades especiales sobre los elementos del repositorio.
- Enviar nuevas revisiones de forma atómica.

6.1.3. Maven 3

Maven es una herramienta para automatizar la gestión de software escrito en Java, incluyendo compilación, pruebas y despliegue, entre otras tareas. Fue creada dentro del proyecto Jakarta (2002), aunque actualmente el proyecto pertenece a la Apache Software Foundation. Posee una funcionalidad similar a la de *Apache Ant*. La diferencia principal entre ambas es que en *Ant* las acciones a realizar se definen en forma procedural paso por paso, mientras que con *Maven* se declara qué «plugins» se van a utilizar, con qué configuración y con qué dependencias y *Maven* se encarga del orden en el que se utilizan las cosas para lograr el objetivo declarado.

Maven utiliza un POM (Project Object Model) para describir el proyecto software a construir, sus dependencias con otros módulos y componentes externos, así como el orden de construcción de los elementos. Es un archivo basado en un formato XML que se ubica en la raíz del proyecto módulo. Viene con objetivos predefinidos para realizar ciertas tareas tales como la compilación del código y su empaquetado. Una característica clave de *Maven* es que está listo para usar en red. Nos permite publicar fácilmente binarios que incluyen todas las dependencias, de forma que sea simplemente descargar y utilizar. Si se desea utilizar una nueva biblioteca, por ejemplo, sólo tendrá que añadirse a las dependencias (al estilo de un paquete Debian) y dejar que *Maven* las descargue cuando haga falta.

El funcionamiento de *Maven* se basa en el uso de un repositorio a donde ir a buscar las dependencias. Cuando *Maven* sale a buscar y consigue una dependencia la guarda en el repositorio local que es un directorio en la máquina del usuario (normalmente *~/.m2/repository* en sistemas basados en UNIX). Las siguientes veces que necesite esta dependencia la obtendrá del directorio local, haciéndolo mucho más rápido que la primera vez.

Otra gran ventaja de *Maven* es que ayuda a imponer que los proyectos sigan una estructura uniforme. Las opciones de compilación, empaquetado, etc. pueden hacerse comunes a todos los proyectos, dejando una configuración mucho más consistente e independiente de los detalles de cada proyecto. *Maven* también puede generar proyectos para *Eclipse* y *NetBeans*, con lo que sólo tenemos que mantener un sistema.

Otro aspecto interesante de *Maven* es que, a partir de la información del POM y del código fuente, proporciona al usuario información de los proyectos que puede llegar a ser muy útil: árboles de dependencias, listas de direcciones, informes de las pruebas, referencias cruzadas entre las fuentes, etc. Asimismo, proporciona guías de buenas prácticas para el desarrollador.

6.1.4. Tycho

Tycho [15] está destinado para la utilización de *Maven* para la construcción de plugins para *Eclipse*, sitios de actualización, aplicaciones RCP y paquetes OSGi. Tycho es un conjunto de plugins *Eclipse* y extensiones para la construcción de plugins para *Eclipse* y paquetes OSGi con *Maven*. Los plugins de *Eclipse* y los paquetes OSGI tienen sus propias metadatos para expresar dependencia, ubicaciones de carpetas, etc. que se encuentran normalmente en un fichero *POM*. Tycho utiliza metadatos nativo para plugins de Eclipse y paquetes OSGi y utiliza el POM para configurar y lanzar la construcción, asegurando de que no hay duplicaciones de metadatos entre *POM* y OSGi. Tycho soporta paquetes, fragmentos, features, sitios de actualización y aplicaciones RCP.

6.1.5. Nexus

La distribución de *Maven* por defecto se descarga los artefactos del repositorio principal de *Maven*. Si bien esto a nivel personal es factible, cuando varios desarrolladores se tienen que descargar los mismos jars pesados una y otra vez es un despilfarro de ancho de banda y de tiempo considerable. Por otro lado, a una organización podría interesarle controlar o restringir de algún modo los artefactos que pueden descargarse los desarrolladores (para que descarguen una misma versión de una biblioteca, para que no usen el repositorio con fines ajenos a la organización, para que sólo empleen dependencias con una licencia compatible a la del proyecto en el que trabajan, etc.). Por ello se suele instalar un gestor repositorio propio, siendo *Nexus* una de las mejores opciones. Entre los motivos por los que debe elegirse Nexus, podemos destacar:

- Puesto que está creado por desarrolladores de *Maven*, la compatibilidad con esta herramienta y la eficiencia en las comunicaciones con su repositorio central son las máximas posibles.
- Es pionero en el formato de repositorio índice.
- Su configuración y mantenimiento son sencillos. Sin embargo, puede ser usado de manera profesional y permite su extensión mediante «plugins».
- Destaca por poseer el modelo de seguridad más robusto y configurable de entre los gestores de repositorios actuales.
- *Nexus* usa Apache Lucene para indexar y buscar en tiempo real, sin necesidad de tener repositorios de contenido o bases de datos.

6.1.6. Sonar

Proyecto que se autodenomina como “una plataforma para la administración de la calidad del código”. Se trata de una herramienta que permite automatizar el análisis del código para gestionar aspectos tales como las nomenclaturas requeridas por una arquitectura o una metodología, el uso de buenas prácticas de programación, la repetición de

código, el porcentaje de código cubierto por pruebas, ciertos parámetros de complejidad de clases y métodos, el porcentaje de código comentado, la evolución de las métricas a lo largo del tiempo, etc. En concreto, la plataforma *Sonar* permite gestionar la calidad del código controlando los siete ejes principales de dicha calidad del código:

- Arquitectura y diseño.
- Duplicaciones.
- Pruebas unitarias.
- Complejidad.
- Errores potenciales.
- Reglas de codificación.
- Comentarios.

6.2. Implementación de los analizadores

Como ya se ha mencionado anteriormente, este proyecto recibe dos tipos de ficheros, los *wsdl* que mediante la herramienta *ServiceAnalyzer* se generan un catálogo el cual deberemos recorrer y por otro lado los ficheros del lenguaje diseñado especialmente para este proyecto, TestSpec. En esta nueva versión de *TestGenerator*, será necesario reescribir el analizador de los ficheros TestSpec de modo que no sea necesario tener que llevar el mantenimiento de dos gramáticas al mismo tiempo.

6.2.1. Analizador de los ficheros TestSpec

En un apartado anterior 5.4 pudimos observar cómo sería la forma de los ficheros *spec*. El formato de estos ficheros está descrito mediante una gramática. Veamos cuáles son las reglas que definen dicha gramática en el listado 6.1.

Una vez que tenemos dicha gramática, tenemos que generar el código del analizador con el cual podemos recorrer los ficheros y crear nuestras estructuras de datos que

guardan las restricciones de los datos que posteriormente vamos a generar de forma aleatoria. Para generar todo esto utilizaremos la herramienta *Xtext*.

XText es una herramienta/marco para el desarrollo de lenguajes DSL textuales externos. Al describir tu propio DSL utilizando el lenguaje de gramática EBNF sencillo de Xtext, el generador creará un analizador, un modelo de meta AST (implementado en EMF) así como un editor de texto completo en Eclipse es decir, cubre todos los aspectos aportando una infraestructura completa al lenguaje, desde los analizadores, proceso de enlazado, compilador o intérprete y, todo ello con plena integración de *Eclipse*.

El marco se integra con la tecnología de modelado de Eclipse como EMF, GMF, M2T y partes de EMFT.

El hecho de que Xtext internamente construye un modelo, hace que la labro de la creación de herramientas alrededor de TestSpec y más allá de TestGenerator sea una tarea más sencilla de lo que en un principio podemos imaginar.

Listado 6.1: Gramática del lenguaje TestSpec

```
Model: (entries+=Entry)*;

Entry: Line ';' | Strategy;

Line: Typedef | VariableList ;

Typedef: 'typedef' (primitive=types | typedef=[Typedef])
              restrictions=Restrictions? name=ID;

VariableList: (primitive=types | typedef=[Typedef])
              vars+=Variable (',' vars+=Variable)*;

Variable: name=ID;

Strategy: 'strategy' name=ID '{' (rules+=Rule)* '}';

Rule: paths+=Path(',' paths+=Path)* ':' strategy=StrategyReference;
```

```
Path: variable=[Variable] ('[' indices+=INTEGER ']')* ('.' field='size')?;

StrategyReference: nameStrategy=ID '(' (listParameter+=StrategyParameter
              (',' listParameter+=StrategyParameter)*)? ')';

StrategyParameter: INTEGER | FLOAT | STRING;

enum types
       : INT = "int"
       | TUPLE = "tuple"
       | LIST = "list"
       | FLOAT = "float"
       | STRING = "string"
       | DATE = "date"
       | DATETIME = "dateTime"
       | TIME = "time"
       | DURATION = "duration"
       ;

Restrictions: '('{Restrictions} (restriction+=Restriction
              (',' restriction+=Restriction)* )? ')';

Restriction: name=ID '=' value=RestrictionValue ;

RestrictionValue: int=INTEGER
       | string=STRING
       | primitive=types
       | typedef=[Typedef]
       | float=FLOAT
   | {RestrictionValue}'{' (list+=RestrictionValue
              (',' list+=RestrictionValue)* )? '}'
   ;
```

```
terminal INTEGER
       : ('+'|'-')?('0'..'9')+ ;

terminal FLOAT
    :   ('+'|'-')?('0'..'9')+ '.' ('0'..'9')* EXPONENT?
    |   '.' ('0'..'9')+ EXPONENT?
    |   ('+'|'-')?('0'..'9')+ EXPONENT
    ;

terminal EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;

terminal STRING :
  '"' ( '\\'('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\'|'0') | !('\\'|'"') )* '"' |
  "'" ( '\\'('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\'|'0') | !('\\'|"'") )* "'";
```

6.3. Implementación del editor

Como ya se ha mencionado en más de una ocasión, para poder generar nuestro editor, se ha usado la herramiento *Xtext*. *Xtext* nos realiza de forma automática, una vez diseñada la gramática, las comprobaciones léxicas y sintácticas. Además nos proporciona una serie de clases y métodos para poder realizar nuestras comprobaciones semánticas. En el listado 6.2 podemos ver un ejemplo de una comprobación semántica.

Trabajar usando Tycho y Nexus ha sido una tarea más dificil de lo que se podría pensar. Para que quede más claro vamos a hablar primero de las partes en las que podríamos dividir los ámbitos de trabajo

- Por un lado tenemos el núcleo de la aplicación en el cual tenemos los analizadores, generadores de datos y formateadores.
- Por otro lado tenemos el editor, un plugin para *Eclipse* el cual nos aporta todas las comodidades para poder trabajar con ficheros *spec*

Listado 6.2: Comprobación semántica del editor para TestSpec

```
@Check
public void checkRestrictionIsUnique(Restriction r) {

	EList<Restriction> superRestriction = ((Restrictions) r.eContainer())
			.getRestriction();
	int contName = 0;
	for (Restriction other : superRestriction) {
		if (r.getName().equals(other.getName()))
			++contName;
		if (contName >= 2) {
			error("Restriction names have to be unique",
				TestSpecPackage.Literals.RESTRICTION__NAME);
			return;
		}
	}
}
```

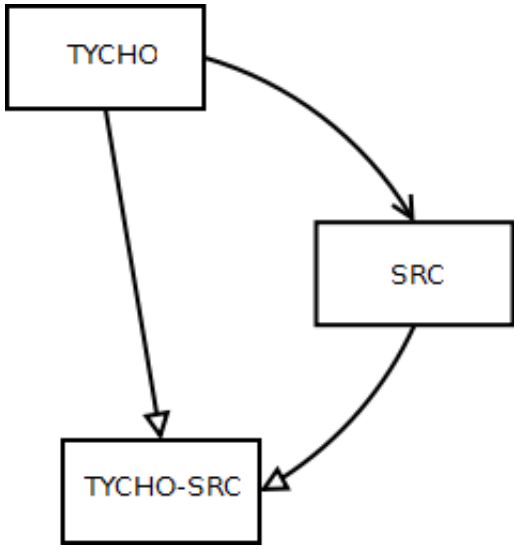

Figura 6.1.: Diagrama de la estructura de directorios

- Por último necesitamos otro plugin en combinación con el anterior para poder comunicar ambas partes anteriores de modo que sea posible generar una salida directamente desde eclipse sin tener que pasar por la terminal.

Al principio se intentó hacer por un lado el núcleo de la aplicación y por otro todo lo referente a *Eclipse* pero esto no era posible puesto que se creaban dependencias cíclicas entre los paquetes, la solución fue la siguiente: crear tres espacios de trabajos distintos uno para cada función

- src: en este directorio tenemos todo lo referente al núcleo de la aplicación.
- tycho: encontramo todo los propio de *Xtext*.
- tycho-src: establecemos la comunicación entre los anteriores.

En el diagrama 6.1 podemos ver la dependencia que existen entre ellos.

6.4. Generadores de datos aleatorios

En esta apartado, veremos cómo hemos generado los datos de forma aleatoria. En el PFC de la técnica, para poder generar dichos datos, nos basábamos en la clase *Random* del paquete *java.util* y se encuentra disponible desde la versión JDK 1.0.

Dicho generador aleatorio, se ha demostrado que genera los número con dudosa calidad, en la cita [16] podemos ver un patrón en la imagen que demuestra la baja calidad de generación que tiene la clase *Random*.

En pos de mejorar la generación de datos, se ha sustituido la clase *Random* del paquete *java.util* por la biblioteca *Uncommons Maths* [17].

La biblioteca *Uncommons Maths* implementa varios PRNG (pseudorandom number generator) y otras distribuciones de probabilidad más allá de la uniforme. Dicha biblioteca recomienda usar Mersenne Twister como PRNG de propósito general, y además la clase hereda de Random, de forma que es un reemplazo limpio del PRNG de Java. Según el autor, supera la batería estándar de pruebas Diehard [18] para PRNG (la *Random* de Java no las supera).

6.5. Estrategias de generación

Para este proyecto se han generado dos tipos de estrategias de generación, la generación de datos según una distribución estadistica y estrategias combinatorias. En los siguientes apartados se detallarán dichas cómo se implementan dichas estrategias.

6.5.1. Distribuciones estadísticas

Las clases que vamos a usar para nuestras estrategias de generación son las siguiente:

- BinomialGenerator
- ExponentialGenerator
- GaussianGenerator

- PoissonGenerator

Cada una de estas clases las usaremos para generar los valores de forma aleatoria que necesitamos según la estrategia en cuestión. Todas las clases perteneces a la biblioteca *Uncommons Maths* la cual nos facilitará la generación de datos.

6.5.2. Estrategias combinatorias

Sum

Dicho algoritmo lo que hace es que dado un número fijado que deba dar como resultado la suma, genera de forma aleatoria distintos valores que cumplen dicho resultado. Por ejemplo: si el resultado de la suma debe de ser 4 y tenemos dos varibales, las posibles soluciones son las combinaciones [0,4], [1,3], [2,2], [3,1] y [4,0]. Para generar los datos usamos el siguiente algoritmo:

1. Generamos tantos número aleatorios en el intervalo [0,1] como variables tengamos menos 1.
2. La suma total de esos número aleatorios no puede ser superior a 1.
3. Multiplacmos el valor que deseamos que de la suma por el primer número generador entre [0,1] y ese es el valor de la primera variable
4. Repetimos el proceso pero sustituyendo el número de la suma total menos el número generado.

Para que quede más claro lo vemos en un ejemplo: Si tenemos 5 variables debemos generar 4 números de forma aleatoria entre [0,1], supongamos que obtenemos los valores 0'2, 0'3, 0'5 y 0'4. El reparto quedaría de la siguiente forma:

- Primera variable: 10 * 0'2 = 2, quedan 10 - 2 = 8.
- Segunda variable: 8 * 0'3 = 2'4, quedan 8 - 2'4 = 5'6
- Tercera variable: 5'6 * 0'5 = 2'8, quedan 5'6 - 2'8 = 2'8

- Cuarta variable: 2'8 * 0'4 = 0'92, quedan 2'8 - 0'92 = 1'78
- Quinta y última variable: 1'78.

EachChoice

La idea básica detrás de la estrategia de combinación de cada opción es incluir a cada valor de cada parámetro en al menos un caso de prueba. Esto se consigue mediante la generación de casos de prueba mediante la selección de valores no utilizados sucesivamente para cada parámetro. Por lo tanto, el algoritmo es muy simple: mientras que hay valores no visitados, una nueva combinación se construye. Si un conjunto tiene cualquier valor no utilizado, éste se inserta en la combinación, de lo contrario, se recoge un valor aleatorio.

Para generar este algoritma lo que hacemos es ir generando un valor cada vez de cada una de las variables, si alguna de ella ya se ha incluido en el caso de prueba, escogemos uno al azar hasta que consigamos que aparezca cada vez un valor distinto de cada dato.

Comb

El algotitmo Comb (en español llamado Peine) se basa en generar primero todas las posibles combinaciones existentes y enumerarlas. A continuación, realiza la selección de las combinaciones más distintas existentes en función de la forma que han sido enumeradas.

Para realizar dicho algoritmo nos hemos basados en el algoritmo del listado 6.3 que podemos ver en la bibliografía [19]

6.6. Representación de los tipos de datos

6.6.1. Enteros

De forma inmediata, podemos pensar que la mejor forma de representar tipos enteros, debería ser generando de forma aleatoria instancias del tipo int de Java. En realidad así

Listado 6.3: Algoritmo Comb

```
result : Set(Combinations)=emptySet
    last : int = getMaxNumberOfCombinations()-1
        // The 1st combination is added
    result.add(getCombination(0))
        // Also the last combination
    result.add(getCombination(last))

    p=(last+1)/2
        // The middle also is added
    result.add(getCombination(p))

    while (|result|<numberOfDesiredCombinations) {
                // k moves by the middle point of each interval
                k : float = p/2
                counter : int = 2
                while (k<last && |result|<numberOfDesiredCombinations) {
                        c : Combination = getCombination(k)
                                if (c! inresult)
                                        result.add(c)
                                // when k is multiplied by counter, it advances
                                // to the next interval
                                k = counter *(p/2)
                                counter = counter +1
                        }
        p = p /2
    }
    return result
}
```

fue como se hizo en un primer lugar hasta que viendo los ficheros *wsdl* que tiene el grupo, pude observar que quería representar el tipo unsigned int, este tipo toma valores en el intervalo [0, 4.294.967.295] este rango no concuerda con el tipo int de Java que sus valores van en el intervalo [-2.147.483.648, 2.147.483.647]. Como solución a dicho problema decidimos usar la clase *BigInteger* localizada en el paquete *java.math*, se trata de una clase que representa enteros de precisión arbitraria.

Para generar un *BigInteger* aleatorio, podemos usar unos de sus constructores que los genera de forma aleatoria, pero esto no es tan sencillo puesto que el constructor recibe el número de bit que necesita para representar dicho número. Con esto hay que tener cuidado, ya que podemos salirnos del rango en el cual lo necesitamos. Para generarlo hacemos lo siguiente:

1. Calculamos la diferencia del intervalo, valor máximo permitido menos el valor mínimo permitido.

2. Generamos el número de forma aleatoria con el número de bits de la diferencia (máximo-mínimo).

3. Sumamos el valor mínimo al número aleatorio resultante.

4. Comprobamos que el número no es mayor que máximo. Si fuese así, volvemos al paso 2.

Veámoslo mejor con un ejemplo. Supongamos que tenemos que generar un número aleatorio en el rango [20, 50]: pasaremos a ejecutar los pasos anteriormente descritos. Calculamos la diferencia del intervalo y nos obtenemos como resultado el número 30 (50- 20 = 30).

Para representar el número 30 son necesarios 5 bits ($2^5 = 32$), por lo que generamos números de forma aleatoria con 5 bits, lo que nos da la posibilidad de representar 32 números. Supongamos que de forma aleatoria se genera el número 13, pasaríamos a sumárselo a la cota inferior del intervalo obteniendo el número 33 (20 + 13 = 33) por lo que este sería nuestro número generado aleatoriamente.

Si por el contrario hubiésemos generado de forma aleatoria el número 31, al sumárselo a la cota inferior del intervalo obtendríamos el número 51 (20 + 31 = 51). Dicho número supera la cota superior del intervalo por lo que tendríamos que generar otro número aleatoria hasta que no superemos el intervalo.

6.6.2. Números en coma flotante

De la misma forma que podemos pensar que para representar un entero podíamos usar el tipo int de Java, podemos pensar que para representar los números de coma flotante, podemos usar el tipo float de Java. Esto no es así ya que esto enmascara un problema un tanto difícil de encontrar y es la forma de representarlo. Los tipos float y double de representan los números en coma flotante en base 2 y no en base 10, esto es así por cuestiones de eficiencia ya que el procesador le es más fácil trabajar con este tipo de representación, Java usa el estándar IEEE 754 para representar dichos números. Esto hace que algunos números no se puedan representar de forma exacta, por ejemplo, el número 0.1 en base 10 es exacto, pero periódico en base 2. Es como intentar representar 1/3 en base 10 resulta un número periódico (0'333333...).

Para resolver este problema, debemos usar un tipo que represente los números en formato decimal: *BigDecimal*, del paquete *java.math*. Este tipo está recomendado por ejemplo para operaciones que impliquen suma de dinero. El problema es que como la representación de cada sumando es inexacta, el resultado acumula los errores de cada uno de esos sumandos. Eso quiere decir que al final, debido al error acumulado, el resultado puede no tener las cotas de exactitud que necesitamos. Internamente *BigDecimal* es un *BigInteger* con una escala determinada que indica la posición del separador decimal.

Para calcular un número en coma flotante de forma aleatoria usamos la fórmula `(max - min) * rnd.nextFloat() + min`, siendo `max` el valor máximo permitido, `min` el valor mínimo permitido y `rnd.nextFloat()` un número calculado de forma aleatoria entre $0{,}0$ y $1{,}0$.

6.6.3. Cadenas

Para generar cadenas de caracteres de forma aleatoria hemos usado la clase *string* de Java para su representación. Podemos generarlo usando una expresión regular. Para ello, nos hemos servido de *Xeger*.

Xeger es un programa que es capaz de generar una instancia de forma aleatoria a partir de una expresión regular. Dicho programa se encuentra bajo la licencia Apache License 2.0 al igual que este proyecto, con lo cual no tenemos ningún problema en utilizarlo.

Xeger es capaz de generar una cadena alfanumérica de forma aleatoria usando un autómata finito no determinista. Dicho autómata representa la expresión regular y en vez de dada una entrada recorrer el autómata para ver si llega a un estado final y validarlo, lo que hace es que va pasando de estado a estado de forma aleatoria hasta llegar a un estado de aceptación y devuelve la cadena que ha generado.

Para generar una cadena alfanumérica aleatoria basta con pasarle la expresión regular `[A-Za-Z0-9]{0,longitud}`, donde `longitud` representa la máxima longitud que pueda tomar la cadena.

En la figura 6.2 podemos observar un autómata finito para explicar mejor el funcionamiento de *Xeger* con un ejemplo. Este autómata representa la expresión regular `a[aeo]*`. *Xeger* actuaria de la siguiente forma: entraría por el estado inicial y pasaría al estado q_1, ahora de forma aleatoria decidiría si realizar una transición al estado q_2, q_3 o al propio q_1 o por el contrario, tomarlo como salida válida. De la misma forma seguiría actuando hasta que de forma aleatoria decida tomar un estado final.

6.6.4. Fechas

Para representar las fechas, hemos utilizado el tipo Java *XMLGregorianCalendar*. Este tipo es el más interesante para representar nuestras fechas, puesto que se corresponde con la sintaxis que utiliza XMLSchema, aunque para poder generar fechas de forma aleatoria hemos utilizado la clase *Calendar*.

El proceso consiste en tomar nuestras fechas máximas y mínimas representadas por instancias de la clase *XMLGregorianCalendar* y pasarla a la clase *Calendar*, esto es así

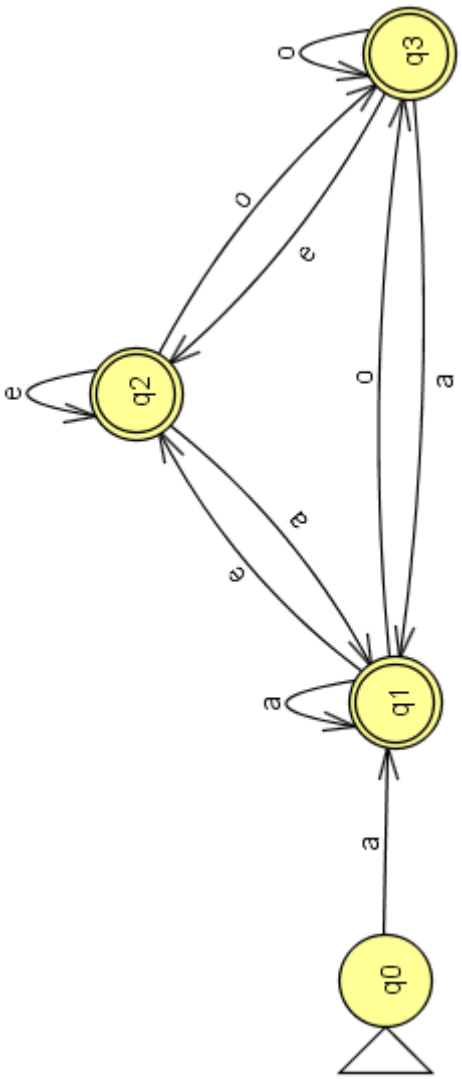

Figura 6.2.: Ejemplo de automata finito generado por Xeger para la expresión regular a[aeo]*

porque la clase *Calendar* tiene un método que nos permite representar la fecha en milésimas de segundo con el método *getTimeInMillis*, con esto lo que hacemos es generar un número aleatorio comprendido entre los milésimas de segundo de la fecha mínima y la fecha máxima y a continuación volver a convertir ese número generado aleatoriamente al tipo *Calendar* y este a su vez al tipo *XMLGregorianCalendar*.

6.6.5. Contenedores

Para generar las clases contenedoras de forma aleatoria, basta con generar el número de objetos que pueda albergar de forma aleatoria del tipo correspondiente.

6.7. Pruebas

Una de las partes fundamentales en cualquier aplicación es el proceso de pruebas. Esto permite aumentar la calidad del sistema reduciendo el número de errores. También permite disminuir la probabilidad de que el sistema falle después de realizar alguna modificación. Existen distintos tipos de pruebas:

- Pruebas unitarias: son aquellas pruebas diseñadas para probar partes muy específicas de la aplicación. Comprueban de forma automática de un conjunto reducido y cohesivo de clases. Estas pruebas son diseñadas por los programadores.
- Pruebas de aceptación: están destinadas a comprobar la funcionalidad del sistema, es decir verifican si cumplen las expectativas pedidas por el cliente.
- Pruebas de integración: son aquellas pruebas que verifican que el sistema funciona de la forma esperada dentro del marco del sistema. En este proyecto, estas pruebas tienen un grado alto de importancia, ya que dicho programa se utilizará dentro del grupo UCASE y deberá funcionar de la forma esperada aunque cambie algún programa del cual depende este proyecto.
- Pruebas de implantación: desde el inicio de un proyecto, el sistema debe de implantarse en un entorno similar al real, posiblemente a menor escala, y compro-

barse su correcto funcionamiento. Este tipo de prueba es muy fácil de seguir en el grupo UCASE debido a su sistema de integración continua.

6.7.1. Naturaleza de las pruebas

La totalidad de las pruebas usadas son pruebas de caja negra, por ser fáciles de mantener a pesar de cambios en la implementación.

Recordemos que las pruebas de caja negra son aquellas que se centran en lo que se espera de un módulo, es decir, intentan encontrar casos en los que el módulo no se atiene a su especificación. Para ello, se apoyan en la especificación de requisitos del módulo, por lo que también se denominan pruebas funcionales. Conociendo una función específica para la que fue diseñado el producto, se pueden diseñar pruebas que demuestren que cada función está bien resuelta. El probador se limita a suministrarle datos como entrada y estudiar la salida, sin preocuparse de lo que pueda estar haciendo el módulo por dentro.

Las pruebas de caja negra son útiles en cualquier módulo del sistema pero están especialmente indicadas en aquellos que van a servir de interfaz con el usuario.

El problema con las pruebas de caja negra no suele estar en el número de funciones proporcionadas por el módulo (que siempre es un número muy limitado en diseños razonables), sino en los datos que se le pasan a estas funciones. El conjunto de datos posibles suele ser muy amplio.

A la vista de los requisitos de un módulo, se sigue una técnica algebraica conocida como “clases de equivalencia”. Esta técnica trata cada parámetro como un modelo algebraico donde unos datos son equivalentes a otros. Si logramos transformar un rango excesivamente amplio de posibles valores reales en un conjunto reducido de clases de equivalencia, entonces es suficiente probar un caso de cada clase, pues los demás datos de la misma clase son equivalentes. Es importante identificar qué rangos de datos pueden alterar el comportamiento del programa y así definir zonas de trabajo. Es imprescindible pasar pruebas con al menos un dato de cada zona, tanto si el programa debe funcionar como si debe dar un mensaje de error. La experiencia indica, además,

que suelen producirse fallos en los bordes de las zonas, por lo que se recomienda probar siempre con datos extremos.

Esta técnica es en la que nos hemos basado a la hora de elaborar la mayoría de las pruebas del sistema, pero necesitamos otro tipo de prueba. El problema está en probar las estrategias, no nos sirve sólo con saber si el dato generado es válido sino que también cómo se ha generado si ha llamado a las clases que debería de llamar y para ello hemos tenido que realizar pruebas de comportamiento para demostrar que se generaban los datos de la forma adecuada.

Los casos de prueba se han definido utilizando el framework *JUnit*, el conjunto de bibliotecas creadas por Erich Gamma y Kent Beck para hacer pruebas unitarias de aplicaciones Java. Para las pruebas de comportamiento, se ha usado el framework *Mockito* [20].

6.7.2. Diseño de las pruebas

Dada a la forma que está estructurada la aplicación, podemos englobar las pruebas unitarias a tres elementos a probar:

- Analizadores: en este punto hay que comprobar que el programa crea las estructuras de datos adecuadas, guardando las restricciones que le corresponde a cada dato. Esto hay que comprobarlo por dos partes, una para los ficheros *wsdl* y otra para los ficheros *spec*.

- Generador: en este parte del programa hay que comprobar que los datos que se generan aleatoriamente cumplen con las restricciones impuestas por la estructuras de datos que almacenan la información, comprobar esto es un tanto difícil puesto que los datos se generan de forma aleatoria así que hay que intentar acotarlos en puntos críticos y jugar con generarlos un número de veces apropiada a cada situación. Además de lo anteriormente dicho, hay que realizar pruebas para comprobar que realmente se generan los datos de la forma deseada, para ello se realizan pruebas de comportamiento usando el framework *Mockito*. También se ha añadi-

do test de Chi-cuadrado para la comprobación que los datos se generán siguiendo una distribución uniforme al ser esta la distribución más personalizada, el resto de las distribuciones no se comprueba ya que la biblioteca elegida contiene dichas pruebas.

- Formateadores: aquí comprobaremos si el formateo de la salida es el adecuado según el dato aleatorio generado. Hay que comprobarlo tanto para el formato CSV como para el de Apache Velocity.

Además de diseñar estas pruebas unitarias, es necesario diseñar pruebas de integración, con ella se pretende cubrir su correcta integración con *ServiceAnalyzer*, para ello hemos creado pruebas con los siguientes *wsdl* que se usan actualmente en el grupo:

- LoanApprovalRPC
- LoanStructured
- MetaSearch
- SquaresSum
- TacService
- LoanApprovalExtended
- MarketPlace
- ShippingServiceAsynchronous

Aparte de estas pruebas, también se han diseñado dos pruebas más de integración, con estas pruebas no se comprueba el resultado sino que se comprueba que el programa sigue su recorrido normal y no lanza ningún tipo de excepción al ejecutarse. Una de las prueba utiliza un fichero wsdl y el formato CSV y la otra utiliza un fichero *spec* y el formato Apache Velocity.

6.8. Validación

En esta fase se ha mejorado la calidad del código. Para mejorar la calidad del código se ha utilizado la herramienta *Sonar*. Dicha herramienta se encuentra alojada en los servidores que usa el grupo UCASE. Al subirse al repositorio una versión nueva del código, *Sonar* comprueba la calidad del mismo.

Sonar es capaz de detectar puntos débiles de nuestro proyecto como errores potenciales en el código, escasez de comentarios, clases demasiado complejas, escasez de cobertura de las pruebas unitarias, etc.

Lo primero que llama la atención es la sección de "Violations" que nos indica los errores que tiene nuestro código dividido en niveles de gravedad. Esta es una visión muy útil para asegurar que nuestro código está escrito de acuerdo a las buenas prácticas de Java mejorando así en eficiencia, usabilidad y mantenibilidad, fundamentalmente.

Sonar también da información del resultado de los test y de su cobertura; así como del porcentaje de líneas que son comentarios y de líneas duplicadas en el código. Este último dato nos puede servir para darnos cuenta de las zonas de la aplicación que están repetidas y que convendría refactorizar en una única clase.

Entre las correcciones realizadas a partir de las indicaciones Sonar podemos citar las siguientes:

- Uso de *StringBuilder*: esta clase se ha empleado en el generador de plantillas para optimizar las concatenaciones. Un *String* es un objeto inmutable, por lo que con cada concatenación estamos creando un objeto nuevo. Si bien en pocos objetos no es importante, a la hora de trabajar con muchos *String*, por ejemplo, cuando la concatenación se produce dentro de un bucle, supondría un despilfarro de memoria. Java provee soporte especial para la concatenación de *String*s con la clase *StringBuilder*. Un objeto *StringBuilder* es una secuencia de caracteres mutable: su contenido y capacidad puede cambiar en cualquier momento.

- Complejidad ciclomática: con esto podemos detectar que métodos son demasiados complejos, si son demasiado complejos probablemente serán más difíciles de

mantener, esto sugiere realizar una refactorización del mismo en la manera de lo posible siempre que el código no pierda su claridad.

- Número mágico: al usar ciertos números, *Sonar* interpreta que pueden tener un significado especial por lo cual sugiere llevarlo al uso de una constante que lo representa por si en un futuro dicho valor cambia.
- Corte de relaciones cíclicas: había dos paquetes con dependencias cíclicas, por lo que se movieron las clases conflictivas de un paquete al otro.

Capítulo 7

Conclusiones

7.1. Valoración personal

La elaboración de este proyecto ha supuesto una gran aportación a mis aptitudes y actitudes como ingeniero. Con este proyecto he realizado una primera toma de contacto con la clase de trabajo que tendré que afrontar en una futura vida laboral.

Una de los valores añadidos a mi persona que ha tenido desarrollar este proyecto, ha sido la colaboración con un grupo de investigación. Dicha experiencia ha sido muy enriquecedora puesto que no sólo he aprendido el uso de ciertas tecnologías, además de ello he podido comprobar cómo se trabaja en grupo asistiendo a distintos seminarios durante el desarrollo del proyecto. Esto ha facilitado aumentar mis competencias transversales como son el trabajo en equipo, expresión oral y escrita así como intervenciones en público.

Por primera vez he trabajado con un entorno de integración continua, valorando su utilidad y la importancia de poder acceder a la versión más reciente de código, sobre todo cuando existe dependencias entre distintos proyectos dentro del mismo grupo. De esta forma he conocido y aprendido a utilizar la herramienta *Maven* para gestión de proyectos, valorar y utilizar las herramientas de control de versiones como puede ser *Subversion*, a valorar la importancia de tener un código de calidad gracias a la herramienta *Sonar*, etc.

Todo esto también me ha servido para valorarme a mí mismo como ingeniero, ya que he estado utilizando herramientas las cuales eran en un principio totalmente desconocidas a mi persona por lo que he tenido que usar distintos manuales, artículos y otras fuentes para poder aprender a usar dichas herramientas. En cierta manera me ha ayudado a perderle un poco el miedo al mundo laboral por el desconocimiento que tengo del mismo.

A través del proyecto, he aprendido a valorar la importancia que tiene realizar una buena batería de pruebas para obtener códigos más fiables y seguros gracias al uso de *JUnit* o *Mockito*.

Cabe mencionar también el grado de comprensión que gracias a este proyecto he logrado alcanzar referente a los traductores y compiladores del lenguaje, puesto que en el mismo he tenido que desarrollar una gramática para posteriormente tener que realizar su análisis léxico y sintáctico con la herramienta *Xtext*, esto ha sido uno de los mayores retos de aprendizaje en mi proyecto junto con elaborar plugins para *Eclipse* ya que estos tienen una curva de aprendizaje bastante pronunciada.

LaTeX ha sido un elemento muy importante en la elaboración de la documentación del presente proyecto. Ha sido necesario aprenderlo partiendo de ningún conocimiento previo, gracias a ello he visto su utilidad y el porqué de su uso frente a los tradicionales procesadores de textos.

También decir que se ha realizado una importante labor de documentación (javadocs, manuales) y que se ha decidido que el proyecto sea Software Libre para que pueda resultar de provecho para la comunidad de desarrolladores.

Por último, dar las gracias al grupo de investigación UCASE y a todos sus componente (alumnos y profesores) puesto que me han ayudado en mi aprendizaje como ingeniero.

7.2. Trabajo futuro

Este proyecto va a tener continuidad dentro del marco de trabajo del grupo UCASE, siendo útil incluso para el desarrollo de nuevos proyectos. A medio plazo, se empleará el

lenguaje TestSpec para generadores basados en estrategias más avanzadas (por ejemplo, algoritmos evolutivos).

Gracias a que *Xtext* genera un modelo, las posibilidad de inclusión en distintos proyectos son bastantes amplias. Como posible mejoras futuras sería dotarlo de más estrategias generadoras o realizar un nuevo formato como por ejemplo JSON. Otra punto de mejora podría ser dotarla con posibilidad de integrarla al framework *JUnit*. Las posibilidades son varias y se abren varios hilos referentes a este proyecto.

Apéndice A

Manual de usuario

En este manual hablaremos de los pasos necesarios para poder usar la herramienta, así como poder escribir adecuadamente especificaciones TestSpec.

La herramienta está destinada a personas con conocimientos sobre pruebas unitarias del software.

A.1. Instalación de TestGenerator

Para instalarse la herramienta bajo una distribución *Linux*, concretamente en *Ubuntu 11.10*, hay que seguir los siguientes pasos:

1. Descargarse el archivo *dist.zip* del enlace `http://bit.ly/y0O2Ej`
2. Descomprimimos el archivo en el directorio que deseemos.
3. Añadimos dicho directorio a la variable `path` del sistema operativo. Si por ejemplo la carpeta la hemos descomprimido en el escritorio, introducimos la siguiente orden en una consola:
   `PATH="$PATH:/home/usuario/Escritorio/test-generator-1.0-SNAPSHOT"`
4. Para que los cambios sean permanente, el usuario debería añadir una línea más al final de *.bashrc*, con este aspecto:

```
export PATH=$PATH: /Escritorio/test-generator-1.0-SNAPSHOT
```

5. Por último tendremos que cerrar sesión y volver a entrar para que los cambios surtan efecto.

Ya está la herramienta lista para usarse.

Para instalarnos los plugins para *Eclipse* debemos realizar lo siguiente:

1. Abrimos nuestro *Eclipse*, si no lo poseemos lo podemos descargar de su web `http://www.eclipse.org/downloads/`

2. Luego pulsamo en la pestaña `Help->Install New Software...`

3. En la ventana emergente que nos aparece, pulsamos el botón `Add..` e introducimos la url `https://neptuno.uca.es/dav/eclipse/interim/`

4. Seleccionamos `Test Generator` e instalamos. En la figura A.1 podemos ver la ventana que nos tiene que aparecer.

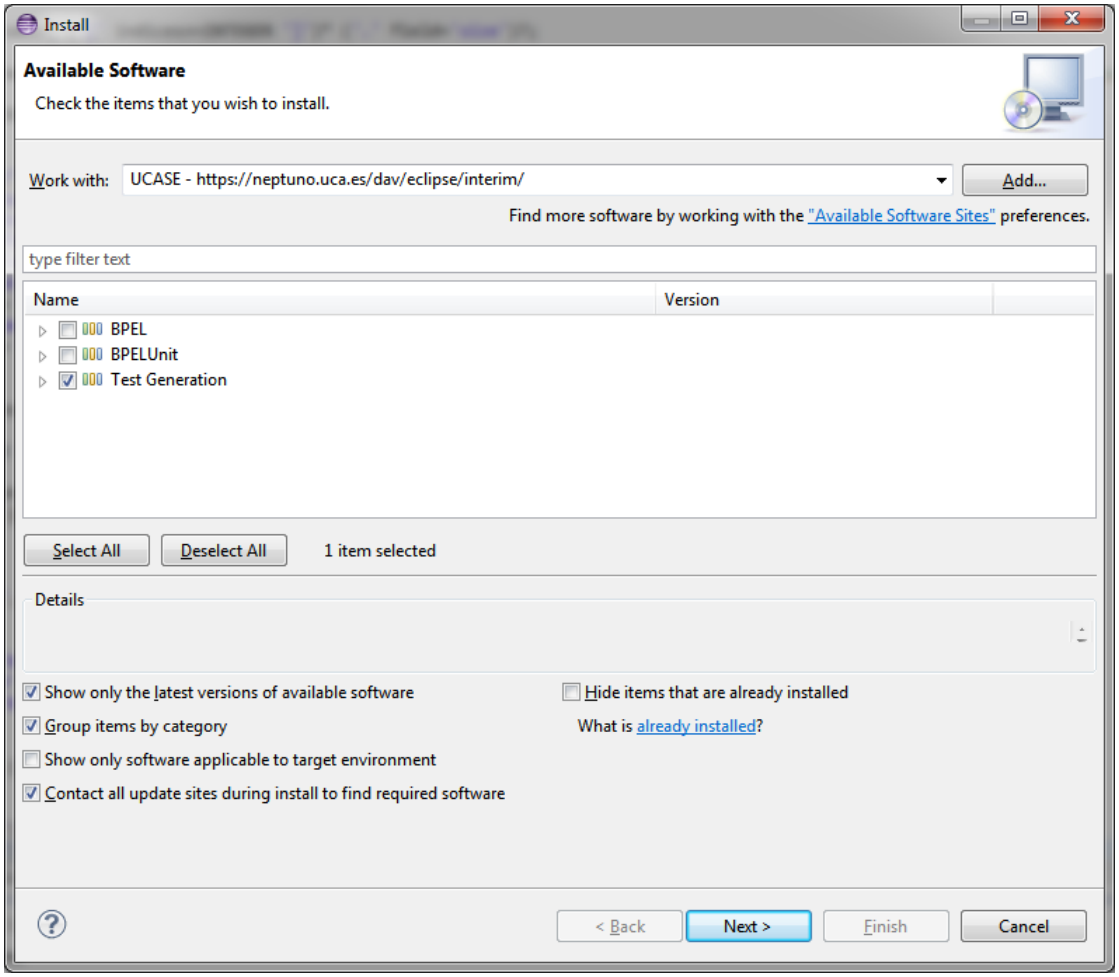

Figura A.1.: Instalación de los plugins

A.2. Uso de la herramienta

A.2.1. Terminal

Para poder usar la herramienta, es necesario abrir una consola y escribir la siguiente sentencia:

```
testgenerator (argumentos)
```

Donde “argumentos” puede tomar una de las siguientes formas:

- `-help`

- `archivo.wsdl servicio operación (in|out|err) [Num test] [-formateador]`

- `archivo.spec [Número de test] [-formateador]`

Si recibe como argumento `-help`, se mostrará la ayuda de la aplicación.

En el caso que reciba un archivo *wsdl*, tendrá que especificar el servicio del archivo, que operación dentro del servicio y elegir el tipo (`in`, `out`, `err`), de forma opcional puede elegir el número de datos aleatorios que desea que genere, por defecto son cinco, y el formato. Para el formato puede elegir entre `-csv` o `-velocity`, por defecto se usará el formato velocity.

Para el caso que reciba un archivo *spec* sólo tendrá que especificar si lo desea, el número de datos a generar y en que formato desea que muestre el resultado.

A.2.2. Plugin de Eclipse

Para poder editar fichero con la extensión *spec* desde *Eclipse*, lo que tenemos que hacer es crearnos un proyecto y añadirles archivos con dicha extensión.

A partir de este momento podemos escribir nuestro ficheros como aparece en la sección A.3.

Vamos a ver algunas de las posibilidades que nos permite este editor:

Con este editor podemos realizar la función de autocompletar, función muy útil cuando no sabemos muy bien que debemos escribir. En la figura A.2 podemos ver un ejemplo.

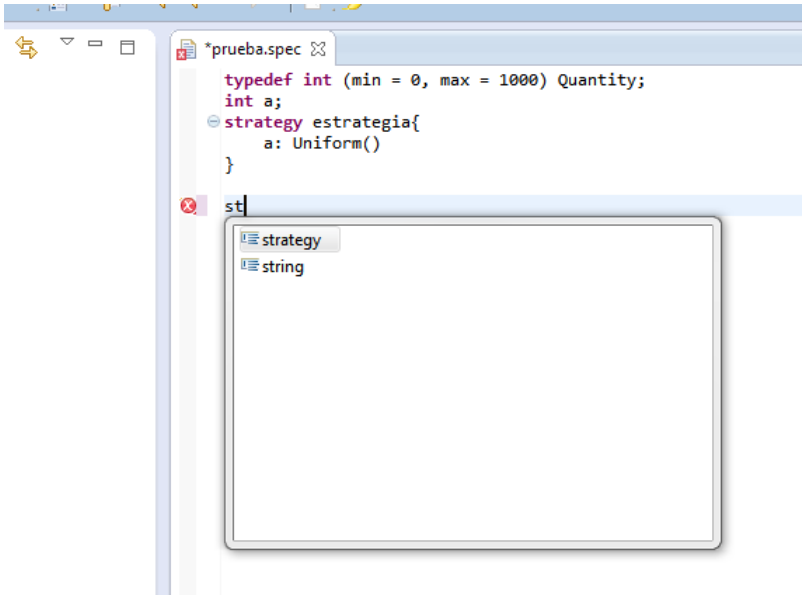

Figura A.2.: Ejemplo autocompletado

Otra de las funcionalidades que nos aporta es la correción de errores léxicos y semánticos. En la figura A.3 podemos ver un ejemplo y el cambio que nos sugiere nuestra aplicación.

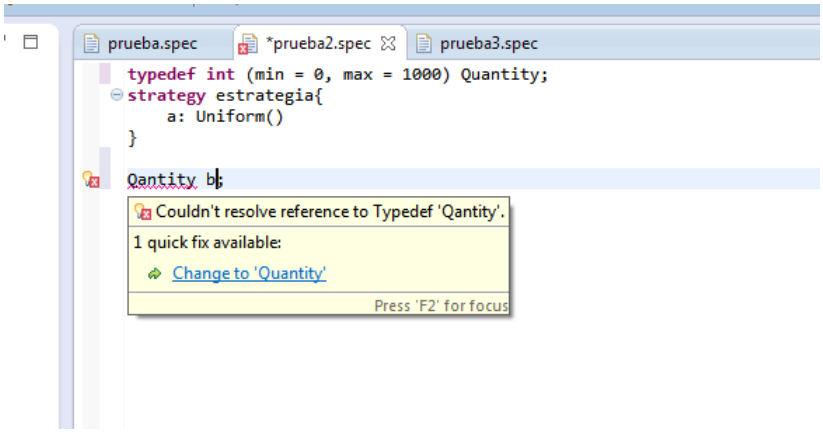

Figura A.3.: Corrección de errores

En la parte derecha de la pantalla, nos aparece la sección «outline», en este apartado

podemos ver bien diferenciadas cada una de las partes de nuestro fichero. Podemos ver un primer bloque llamado `Typedefs` las definiciones que hayamos detallado, como segundo bloque aparece `Declarations` en el cual aparecen nuestras declaraciones y por último aparece el bloque de las estrategias `Strategies` en el que aparecen las que hayamos especificado. En la figura A.4 podemos ver un ejemplo.

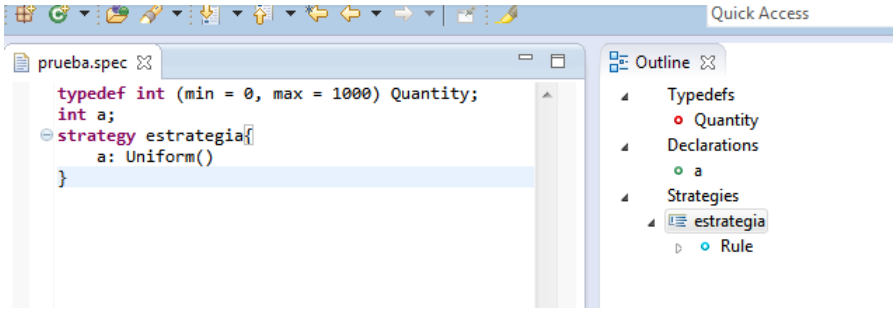

Figura A.4.: Menu outline

Una vez que ya tengamos realizado nuestro fichero, debemos pasar a generar los datos. Para ello pulsamos con el segundo botón del ratón encima del area de edición del fichero y elegimos la opción Generate `Test-Case...` (A.5). A continuación, nos aparece una ventana emergente la cual nos solicita cuantos casos de prueba queremos generar, el tipo de formato de salida, el nombre que queremos darle al fichero y por último el nombre de la estrategia que queremos usar (dicho campo es opcional), en la figura A.6 podemos observarlo. Por último pulsamos sobre el botón aceptar y nos aparecerá un mensaje de éxito si todo ha ido de la manera adecuada(figura A.7). En nuestro proyecto aparecerá una carpeta con el fichero generado, como podemos ver en la figura A.8.

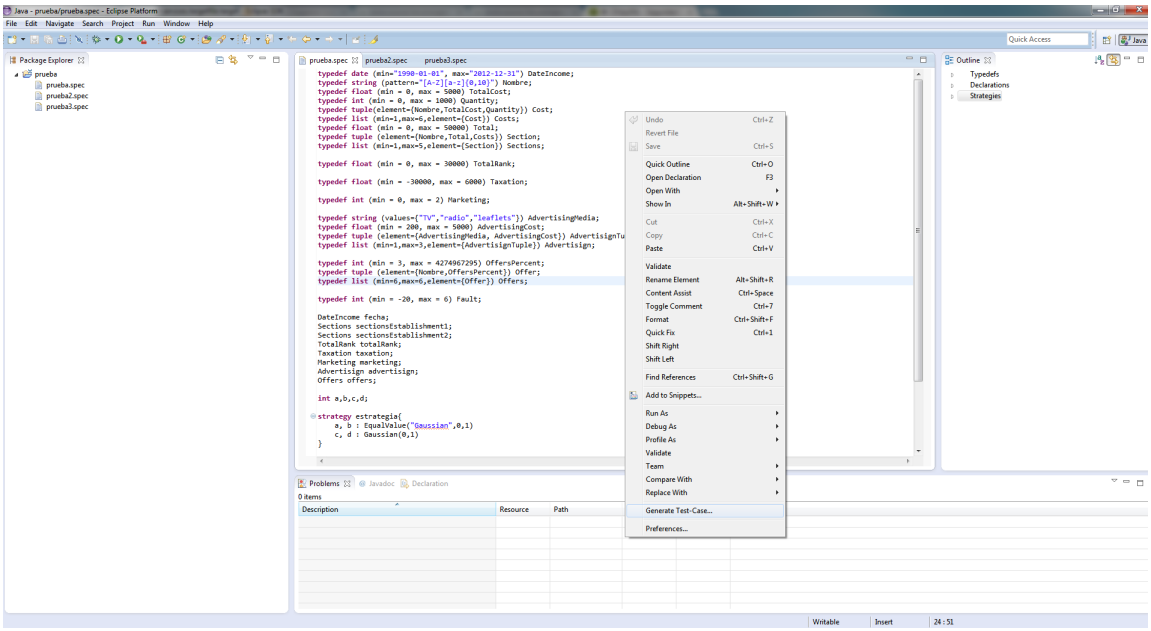

Figura A.5.: Instalación de los plugins

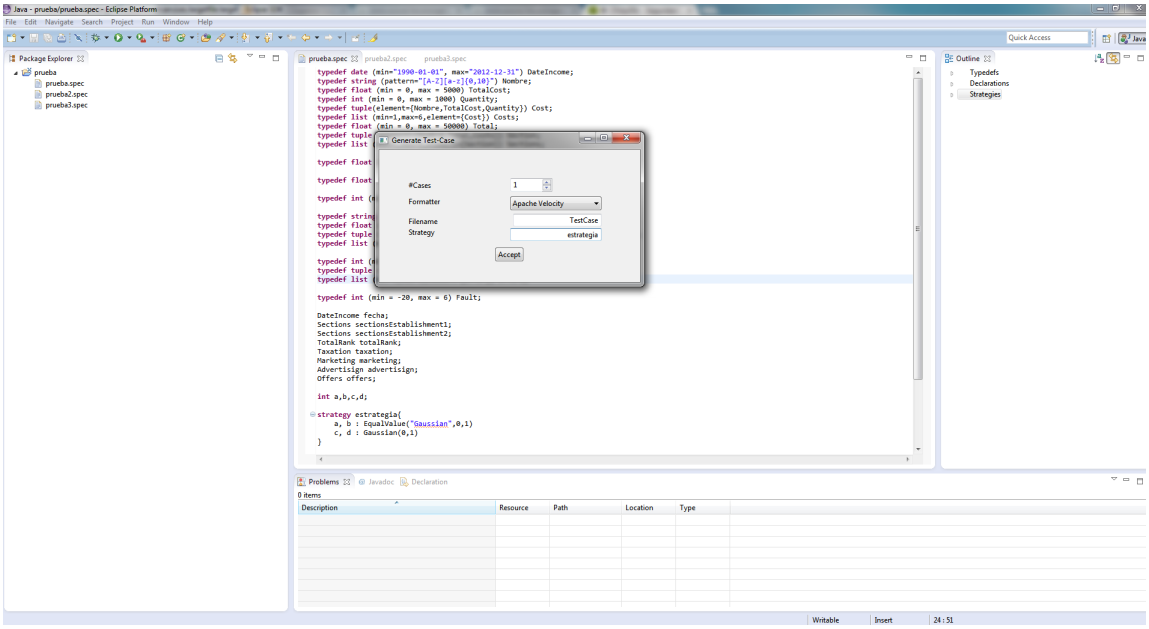

Figura A.6.: Ventana de generación

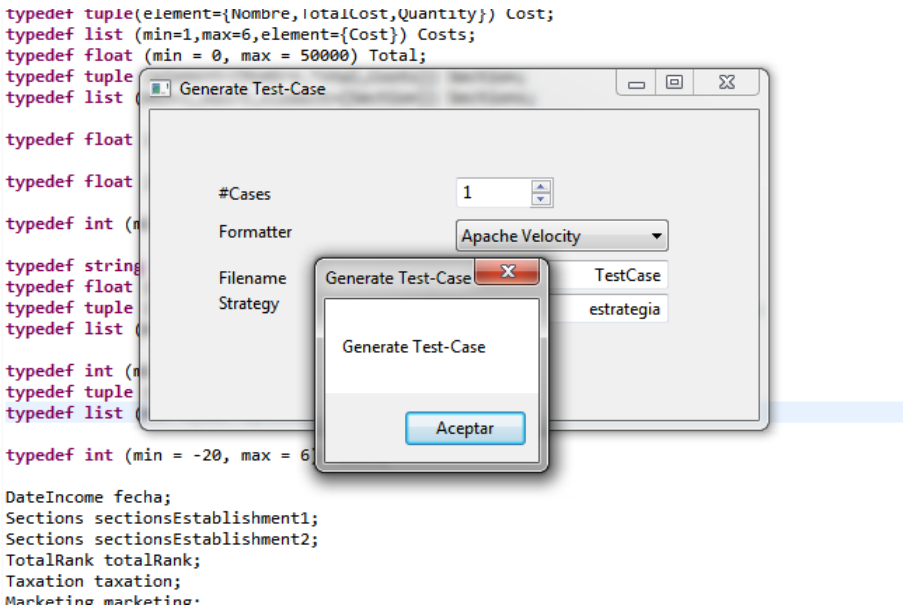

Figura A.7.: Éxito en la generación

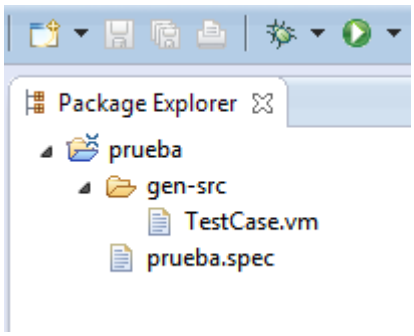

Figura A.8.: Resultado de la generación

A.3. Creación de ficheros TestSpec

TestSpec es el lenguaje especialmente diseñado para realizar especificaciones abstractas del formato de los datos generados por *TestGenerator*.

Una especificación TestSpec tiene tres tipos de sentencias. Mediante el primer tipo se definen los tipos con sus restricciones (sentencias «typedef»). En el segundo, se de-

claran las variables de cada tipo (sentencias «declaration»). En el último se definen las estrategias (sentencias «strategy»).

A.3.1. Sentencias «typedef»

Las sentencias «typedef» vienen estereotipadas de la siguiente forma:

```
typedef Tipo (r1 = valor, r2 = valor...) NombreDefinicion;
```

Dónde:

- typedef: es una palabra reservada de la gramática.
- restricción (r): restricciones que tendrá el tipo anteriormente especificado.
- valor: corresponde con la asignación que tendrá la restricción. Si es única vendrá representado por una sola sentencia, si corresponde con un conjunto de valores aparecerán entre llaves y separadas con comas valor1, valor2...
- NombreDefinicion: identificador asignado al typedef.

Los tipos que podemos definir (sección «typedef») son los siguientes:

- string
- int
- float
- date
- time
- dateTime
- duration
- list
- tuple

A continuación veremos cuáles son las restricciones opcionales u obligatorias que deben tener cada tipo:

- element: esta restricción es obligatoria si el tipo es alguno de los contenedores, representa el tipo de elemento que va a contener. En el caso de las tuplas podríamos necesitar una lista de tipos, por lo que el valor de `element` es una lista ordenada de cadenas separadas por coma.
- min: Este atributo tiene un significado u otro en función del tipo al que se aplique. Aplicado a tipos numéricos, representa el límite inferior inclusivo del espacio de valores del tipo. En el caso de que se aplique a un tipo cadena, indica la longitud mínima que ésta ha de tener. Sin embargo, si es un tipo list, `min` indica el número mínimo de elementos que puede contener la lista.
- max: análogamente a la definición de `min` se le puede aplicar sustituyendo la palabra mínimo por máximo.
- values: Restringe el espacio de valores de un determinado tipo al conjunto de valores especificados.
- pattern: Restringe el espacio de valores de un determinado tipo, restringiendo el espacio léxico a literales que siguen un determinado patrón. El valor debe ser una expresión regular. Esta restricción sólo es aplicable al tipo de dato `string`.
- fractionDigits: controla el número de decimales que deberá contener `float`.
- totalDigits: controla el número total de dígitos que tendrá un número.

Se podrá usar un typedef definido como tipo de un typedef.

A.3.2. Sentencias «declaration»

En las sentencias para realizar declaraciones, aparecerá el nombre de una de las definiciones realizadas en el conjunto de instrucciones «typedef» y el identificador que le vamos a otorgar a la variable, seguido del carácter fin de línea ';'.

Además, se podrá realizar declaraciones de los tipos básicos int, float y string sin que éstos estén definidos en las sentencias «typedef» anteriormente definidas.

A.3.3. Estrategias «strategy»

Las distintas estrategias que podemos usar son las siguientes:

- Uniform
- Gaussian
- Normal
- Poisson
- Exponential
- Sum
- EachChoice
- Comb
- EqualValue

A.4. Ejemplo de un fichero *spec*

Podemos ver un ejemplo en el listado A.1. En el que podemos apreciar en las líneas 1–6 el conjunto de sentencias typedef donde:

- La línea 1 define un tipo cadena denominado boolean que podrá tener como valores las palabras “true” o “false”.
- En la línea 2 podemos ver un typedef con identificador *CaraDado* que es un entero que podrá tener los valores en el rango [1,6].
- La línea 3 es una tupla donde el primer elemento es una cadena y el segundo es del tipo *CaraDado* y este typedef se identificará como *ResultadoTirada*.

- En la línea 4 se define un tipo de lista de enteros con 1 elemento como mínimo.
- En la línea 5 se define un tipo entero con valor mínimo de 0, al que identificaremos como `positiveNumber`.
- En la línea 6 se especializa el tipo `positiveNumber` dándole de valor máximo 100 y llamándole `smallNumber`.

En las sentencias de las declaraciones, podemos ver que se declaran cuatro variables de los tipos “boolean”, “ResultadoTirada”, “ListaNoVaciaInt” y “smallNumber”.

Listado A.1: Ejemplo de especificación TestSpec

```
1  typedef string (values={"true", "false"}) Boolean;
2  typedef int (min=1, max=6) CaraDado;
3  typedef tuple (element = {string, CaraDado}) ResultadoTirada;
4  typedef list (min=1, element = int) ListaNoVaciaInt;
5  typedef int (min=0) PositiveNumber;
6  typedef PositiveNumber (max=100) SmallNumber;
7
8  Boolean myBoolean;
9  ResultadoTirada tirada;
10 ListaNoVaciaInt numeros;
11 SmallNumber myNumber;
12
13 strategy Estrategia{
14        tirada: Gaussian(0,1)
15        myBoolean, myNumber: Comb()
16 }
```

Apéndice B

Manual de desarrollador

En este manual hablaremos de los pasos necesarios que se deberán llevar a cabo si desea descargarse el código fuente y como poder compilarlo.

Este manual está elaborado y probado bajo una distribución *Linux*, concretamente bajo *Ubuntu 11.10*.

B.1. Requisitos del sistema

Para poder obtener el código fuente, debemos tener conexión a Internet en el equipo y tener instalado *Subversion*.

Para instalar *Subversion*, abrimos una terminal y escribimos lo siguiente:

```
sudo apt-get install subversion
```

Los requisitos necesarios para poder compilar el sistema son tener instalado JDK además de *Maven* y *JUnit*, para ellos abrimos una terminal y escribiremos la siguiente orden:

```
sudo apt-get install openjdk-6-jdk maven junit
```

B.2. Obtención del código

Para poder obtener el código fuente del repositorio *SVN*, abrimos una terminal y nos vamos al directorio donde queremos alojar el código. Luego escribimos la siguiente or-

den:

```
svn co https://neptuno.uca.es/svn/sources-fm/trunk/src/test-generator/
```

Con esto obtenemos una copia local de la última versión del código de todos los proyectos del grupo.

Ya es posible navegar por los distintos proyecto, para explorar el código nos dirigiremos a la carpeta *src* dentro de uno de ellos. La estructura de cómo está organizado el código es la siguiente:

main/java Directorio principal que contiene el código Java de la aplicación.

test/java Contiene el código de las pruebas.

main/resources Directorio que contiene los recursos utilizados por el código principal.

test/resources Alberga los recursos necesarios para ejecutar las pruebas.

main/assembly Aquí se encuentra los descriptores de distribuciones.

A la hora de desarrollar, le serán útiles los objetivos predefinidos de *Maven*. A continuación se exponen los principales:

clean Elimina todos los directorios de despliegue del proyecto.

compile Compila el código fuente del proyecto.

deploy Despliega el paquete resultante en un repositorio central para ser compartido.

install Instala el paquete en el repositorio local.

package Crea un paquete con el proyecto a partir de las clases compiladas y recursos.

site Genera un sitio con la documentación del proyecto.

test Ejecuta las pruebas o test del proyecto.

Para ejecutar alguno de estos objetivos, abrimos la terminal y nos situamos en el directorio del proyecto. A continuación escribimos la orden en la terminal de la siguiente forma:

```
mvn objetivo
```

Trabajar directamente desde un editor de texto la manipulación de ficheros Java puede ser una tarea bastante engorrosa. Por ello se recomienda usar un IDE como puede ser *Eclipse* usando algún «plugin» de integración con *Maven*. Estos «plugins» permiten usar *Maven* desde una interfaz más amigable, evitando así la línea de órdenes y contando con las funcionalidades de refactorización y notificación de errores y avisos de compilación en directo que ofrecen el IDE. Las características añadidas a estos entornos son:

- Construir proyectos *Maven* desde el IDE.
- Gestión de dependencias basadas en la sincronización con el pom.xml asociado al proyecto.
- Resolución de dependencias *Maven* en el espacio de trabajo sin necesidad de instalar en repositorios *Maven* locales.
- Descarga automática de las dependencias requeridas desde los repositorios *Maven* remotos.
- Asistentes para crear nuevos proyectos *Maven*, editar los ficheros pom.xml, así como para permitir soporte *Maven* en proyectos ya existentes.
- Búsqueda rápida de dependencias en los repositorios remotos de Maven.
- Avisos en el editor Java para buscar las dependencias o ficheros JAR por el nombre de la clase o del paquete.
- Integración con otras herramientas del IDE de desarrollo elegido.

También necesitamos diferenciar distintos espacios según en la parte del código en la cual queramos desarrollar. Si deseamos trabajar en el núcleo de la aplicación debemos tener un espacio de trabajo con los siguientes proyectos ubicados en el directorio *src*:

- service-analyzer
- test-generator

- test-generator-api
- test-generator-random
- test-generator-xtext
- test-generator-xtext-lib

Si lo que deseamos es trabajar es en el apartado que se encarga de realizar el plugin para *Eclipse* debemos tener un espacio de trabajo con los siguientes proyectos situados bajo el directorio *tycho*:

- es.uca.webservices.testgen.parsers.spec.xtext
- es.uca.webservices.testgen.parsers.spec.xtext.standalone
- es.uca.webservices.testgen.parsers.spec.xtext.tests
- es.uca.webservices.testgen.parsers.spec.xtext.ui

Si queremos trabajar en la integración entre los dos anteriores debemos tener un espacio de trabajo con los siguientes proyecto ubicados en el directorio *tycho-src*

- es.uca.webservices.targetfile
- es.uca.webservices.testgen.parsers.spec.xtext.deps.feature
- es.uca.webservices.testgen.parsers.spec.xtext.feature
- es.uca.webservices.testgen.parsers.spec.xtext.menu
- es.uca.webservices.updatesite

Bibliografía

[1] E. Blanco Muñoz, A. García Domínguez, J. J. Domínguez Jiménez y I. Medina Bulo. Propuesta de una arquitectura para la generación de mutantes de orden superior en WS-BPEL. En *Actas de las XVI Jornadas de Ingeniería del Software y Bases de Datos*, páginas 537–542. A Coruña, España, septiembre 2011.
URL http://www.sistedes.es/jornadas2011/jisbd.htm

[2] J. J. Domínguez Jiménez, A. Estero Botaro, A. García Domínguez y I. Medina-Bulo. GAmera: an automatic mutant generation system for WS-BPEL compositions. En *Proceedings of the 7th IEEE European Conference on Web Services*, páginas 97–106. Eindhoven, Países Bajos, noviembre 2009.

[3] M. n. Pérez Montero, A. García Domínguez y J. J. Domínguez Jiménez. *Generador de casos de prueba aleatorio basado en especificaciones abstractas*. Proyecto fin de carrera, University of Cádiz, Cádiz, España, enero 2012.
URL http://hdl.handle.net/10498/11695

[4] K. Ballinger, C. Ferris, M. Gudgin, C. Kevin Liu, M. Nottingham y P. Yendluri. Basic profile - version 1.1 (Final). Informe técnico, Web Services Interoperability Organization, agosto 2004.
URL http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html

[5] M. Fowler. Domain-specific languages: An introductory example. http://www.

informit.com/articles/article.aspx?p=1592379, septiembre 2010. Fecha de última comprobación: 16 de julio de 2013.

[6] I. Sommerville. *Ingeniería del Software*. Addison-Wesley, sexta edición, 2002.

[7] Pagina oficial de JCheck. http://www.jcheck.org/. Fecha de última comprobación: 1 de diciembre de 2011.

[8] Pagina oficial de QuickCheck. http://java.net/projects/quickcheck/pages/Home. Fecha de última comprobación: 1 de diciembre de 2011.

[9] B. Eckel. *Piensa en Java*. Pearson Education, 2007.

[10] Pagina oficial de JUnit. http://www.junit.org/.

[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad y M. Stal. *Pattern-Oriented Software Architecture: A System Of Patterns*. Wiley, 1996.

[12] Combinatorial testing page. http://alarcosj.esi.uclm.es/CombTestWeb/combinatorial.jsp.

[13] B. P. L. Marco Polo Usaola. A framework and a web implementation for combinatorial testing. Informe técnico, University of Castilla-La Mancha.
URL http://alarcosj.esi.uclm.es/CombTestWeb/stuff/wpCombinatorial.pdf

[14] G. Aburruzaga García, I. Medina Bulo y F. Palomo Lozano. *Fundamentos de C++*. Universidad de Cádiz, 2009.

[15] Pagina oficial de Tycho. http://eclipse.org/tycho/.

[16] Demostración del patrón de generación de datos de la clase Random de Java. http://www.alife.co.uk/nonrandom/.

[17] Pagina oficial de Uncommons Maths. http://maths.uncommons.org/.

[18] Enlace wikipedia de tests de Diehard. http://en.wikipedia.org/wiki/Diehard_tests.

[19] Algoritmo Comb. http://alarcosj.esi.uclm.es/CombTestWeb/algorithms/comb.html.

[20] Pagina oficial de Mockito. https://code.google.com/p/mockito/.

GNU Free Documentation License

Version 1.3, 3 November 2008

<http://fsf.org/>

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to

software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under

this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in

another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally

prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the

"History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes

a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various docu-

ments with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and

disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to

the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-

BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with . . . Texts.” line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.